Computer Principles and Design in Verilog HDL

计算机原理与设计

——Verilog HDL版

李亚民　著

清华大学出版社
北京

内 容 简 介

本教科书讲述计算机原理、计算机设计以及如何用 Verilog HDL 实现设计。主要内容包括：计算机基础知识及性能评价方法；数字电路及 Verilog HDL 简介；计算机加、减、乘、除及开方的各种算法（包括 Wallace Tree 快速乘法器和 Newton-Raphson 及 Goldschmidt 除法和开方算法）及其 Verilog HDL 实现；指令系统结构和 ALU 及多端口寄存器堆的 Verilog HDL 设计；单周期、多周期和流水线 CPU 的 Verilog HDL 设计；精确中断和异常处理及其电路实现；浮点算法及带有浮点部件 FPU 的流水线 CPU 的 Verilog HDL 设计；多线程 CPU 的 Verilog HDL 设计；存储器、Cache 和虚拟存储器管理以及带有 Cache、TLB 和 FPU 的 CPU 设计；多核 CPU 的 Verilog HDL 设计；异步通信接口 UART、PS/2 键盘与鼠标接口、视频图像阵列 VGA 接口、I2C 串行总线接口和 PCI 并行总线接口的 Verilog HDL 设计；高性能计算机及互联网络设计。书中的 Verilog HDL 源代码基本上都附有功能仿真波形，以便加深对计算机原理的理解和对计算机设计方法的掌握。

本书可用作高等院校计算机及信息专业本科生和研究生教材，也可供自学者阅读。

图书在版编目（CIP）数据

计算机原理与设计：Verilog HDL 版/李亚民著. --北京：清华大学出版社，2011.6
ISBN 978-7-302-25109-5

Ⅰ.①计…　Ⅱ.①李…　Ⅲ.①电子计算机－基础理论 ②硬件描述语言，Verilog HDL－程序设计　Ⅳ.①TP3

中国版本图书馆 CIP 数据核字(2011)第 049729 号

责任编辑：薛　慧
责任校对：刘玉霞
责任印制：何　芊

出版发行：清华大学出版社　　**地　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编**：100084
社 总 机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市金元印装有限公司
经　销：全国新华书店
开　本：185×260　　**印张**：33.75　　**字　数**：812 千字
版　次：2011 年 6 月第 1 版　　**印　次**：2011 年 6 月第 1 次印刷
印　数：1～5000
定　价：49.00 元

产品编号：035242-01

序 言

李亚民教授长期从事计算机原理和体系结构的教学与研究，他出版过三本教科书[35, 36, 37]，发表过大量的杂志与会议文章。本书是他近年来执教和科研的实践总结。他的《计算机组成与系统结构》教科书在2000年由清华大学出版社出版；2005年清华大学出版社出版的《CPU芯片逻辑设计技术》是基于李亚民教授在清华大学系列讲座的内容而整理出版的。

李亚民教授的这本新书和上述两本书相比，有很多新的发展和不同点。统观全书，它贯穿了用Verilog HDL来描述计算机的各种部件和用Verilog HDL技术来设计CPU各部件。这是本书一个很重要的特点。这种描述和设计技术是要作者花费很大工夫自己去理解和实践的。单从这点而言，可见本书作者在著写本书时，要投入比著写一般"计算机原理"书籍更多的时间和心血。这个特点带给读者的好处是：学完本书后，不仅懂了计算机原理，也可以基本掌握计算机的一种重要的设计技术。

此外，这本书的内容也增加了一些重要的环节，如CPU中的多核和多线程技术。大家知道，多核和多线程是今后计算机较长期间的发展方向，尤其是近年来高性能计算进入千万亿次时代以后，多核和多线程技术会有更蓬勃的发展。本书这方面的内容也是计算机学生应该掌握的新内容。本书还增加了浮点部件FPU设计、浮点和整数部件结合成整体的完整的流水线CPU设计、带有Cache和TLB的多核CPU的设计、常用I/O接口的设计。本书中所讲授的计算机各部件，都尽量给出了Verilog HDL源代码。这些Verilog HDL源代码都经过Quartus II或者Iverilog的验证。这也体现了作者写书的严谨性和所花费的精力。

中国的计算机事业在改革开放以来，取得了飞速的发展。在高性能计算、网络技术、软件、计算机产业等方面都取得了举世瞩目的成就。然而，我国必须继续重视计算机发展的基础，这就是计算机的设计技术，尤其是计算机芯片设计技术。这方面的不足是和计算机教育分不开的。我国很多大学的"计算机原理"、"计算机组成与结构"的课程讲授和计算机设计技术基础脱离，学生学完计算机原理与组成却一点未掌握计算机设计技术。

本书特点是理论和实践的密切结合。希望本书的出版和推广，能加强大学计算机教学领域内的理论与实践的结合。很多工程领域的结构都好像是个金字塔，只有基础更广泛和更结实了，金字塔的塔尖才可以构建得更高。我们寄希望于广大的大学生和研究生！只有更多的年轻人掌握了计算机和CPU的设计技术，我国的计算机和CPU设计技术才会攀登新的高峰！

李 三 立
中国工程院院士　清华大学教授
2011年5月1日

前 言

欢迎阅读本书。首先郑重声明：本书所载的 Verilog HDL 源代码可以随便使用、随便更改。但本书作者对任何大学、研究所、公司或者个人由于使用本书所载的 Verilog HDL 源代码设计 CPU 或者任何其他电路所造成的任何损失不承担任何责任。

也许有读者会问："现在出版的计算机原理、计算机组成与结构的教科书已经很多了，你为什么还要写这本书?"诚如本书书名《计算机原理与设计 —— Verilog HDL 版》所提示的那样，本书不光讲计算机原理，也讲计算机设计，而且是用 Verilog HDL 讲计算机设计。如果只看教科书讲的原理而不自己动手设计，就如同有人向你描述有一种水果是什么什么味道，听了半天也只能感觉和想象。还不如花一分钟亲口尝一尝：噢，原来是这个味道，这就是传说中的梨子吗?

让读者知道如何自己动手用 Verilog HDL 设计 CPU 及 I/O 接口和总线是作者要写本书的原因之一。本书的一个特点是所有 CPU 和 I/O 接口及总线的 Verilog HDL 源代码全部由作者本人写出并经过 CAD/CAE 软件验证。作者在 2000 年由清华大学出版社出版过一本《计算机组成与系统结构》的教科书[36]，得到广大读者和网友的好评及推荐，有些大学也选其作为课程教材或参考书使用。与那本书不同，这本《计算机原理与设计 —— Verilog HDL 版》给出具体的 CPU、I/O 接口和总线设计的 Verilog HDL 源代码并附有仿真波形，以使读者能加深理解和供读者在设计自己的电路时参考，同时也增加了多线程和多核技术等内容。另一个原因是现在用硬件描述语言来设计电路变得很流行，已成主流，用逻辑图的方法直接设计电路的人越来越少了。但大部分介绍硬件描述语言的书都只停留在简单的数字电路的设计上，因此作者想把它引入到计算机原理的教科书中。

也许有读者还会问："那你为什么选用 Verilog HDL，而不是其他的硬件描述语言呢?"回答这个问题其实有些难。目前比较流行的硬件描述语言主要有 Verilog HDL 和 VHDL 两种。大家都说 Verilog HDL 像 C 语言，而 VHDL 像 C++ (实际上像 Ada 语言)。由于作者对 C 比对 C++ 更熟悉一些，所以开始自学硬件描述语言时就自然而然选了 Verilog HDL。据作者后来有限的调查得知，一些大公司也都使用 Verilog HDL。调查的对象仅限作者指导过的本科生和研究生，他们毕业之后有些人目前在著名的大公司从事集成电路的设计工作。看来作者当初的选择是对的。作者不是说选择 VHDL 就不对，其实用熟练了都一样，都是很好的设计硬件的语言工具。

考虑到初学者的需要，本书从最基本的二进制数的运算和逻辑电路的设计讲起，顺便介绍 Verilog HDL。因此学习本书时不需要任何先修科目，不懂 C 语言也没有关系。"那是不是这本书的内容太过简单了呢?"不是。本书要讲到如何设计一个带有整数部件 (Integer Unit，IU)、浮点部件 (Floating Point Unit，FPU)、指令 Cache (一种特殊的小容量的快速存储器)、数据 Cache、指令 TLB (Translation Lookaside Buffer，一种用于地址转换的快速缓冲区) 和数据 TLB 的多核 CPU。其中的 Cache 完全由硬件控制；TLB 不命中时产生异常，在异常处理程序中访问用于存储器管理的页表 (Page Table)，然后填充 TLB；FPU 能对 IEEE 754 单精度浮点数进行加、减、

乘、除和开方运算：加减法用先行进位加减法器实现、乘法用 Wallace Tree 实现、除法和开方用 Newton-Raphson 算法实现。这些都是实际的 CPU 设计中常用的部件和算法。一般的教科书都是分开讲它们的原理。本书也讲原理，但更重要的是把它们有机地结合在一起，设计出一个完整的 CPU。当然，Verilog HDL 源代码和仿真波形也在书中给出。

稍微讲具体一些。本书的内容包括：计算机基础知识及性能评价；逻辑电路及 Verilog HDL 简介；计算机算法及 Verilog HDL 实现；指令系统及 ALU (Arithmetic Logic Unit) 设计；单周期、多周期和流水线 CPU Verilog HDL 设计；精确中断和异常处理及电路实现；浮点算法及 FPU Verilog HDL 设计；带有 FPU 的流水线 CPU Verilog HDL 设计；多线程 CPU Verilog HDL 设计；存储器和虚拟存储器管理；带有 Cache、TLB 及 FPU 的 CPU 设计；多核 CPU Verilog HDL 设计；输入/输出接口和总线设计；高性能计算机及互联网络设计。更详细的内容就只有看目录了。

除了 Newton-Raphson 算法，本书也介绍 Goldschmidt 除法和开方算法。这两种算法都被著名的公司 (比如 Intel 和 IBM) 在设计 CPU 时使用。输入/输出接口部分除了介绍基本的概念和原理之外，还给出了常用 I/O 接口及总线的具体的设计方法，比如异步通信接口 UART、PS/2 鼠标和键盘接口、VGA 接口、I2C 串行总线和 PCI 并行总线等。除了 PS/2 鼠标，其他的接口和总线都给出了具体的 Verilog HDL 设计实例。PS/2 鼠标讲了原理，设计部分当做习题留给了读者。高性能计算机及互联网络设计部分除了讲述一般的设计方法之外，也描述了由作者提出的新型互联网络，包括 Dual-Cube[15]、Metacube[20] 和 RDN (Recursive Dual-Net)[19]。这些互联网络具有用较少的端口连接更多的 CPU 和存储器板并且保证它们之间的通信距离比较短的优点，非常适合用来构建下一代超大规模的超高性能计算机。

"好像内容太多了，一个学期学不完。" 以上内容可分成两部分，一部分放在本科讲，另一部分给研究生讲。举例来讲，TLB 设计、FPU 设计、多线程和多核 CPU 设计、高性能计算机及互联网络设计等内容可在研究生课程中讲授，其余部分在本科课程中教授。如果在本科阶段能设计出流水线 CPU 就已经很好了，课时充裕的话再加上中断和 Cache 设计。存储器管理要讲，但不一定要设计 TLB。以上只是举例，大家可以根据学校的具体情况做出不同的安排。

"那么需要什么样的环境呢?" 如果只想验证用 Verilog HDL 设计的电路的逻辑功能是否正确，只要有一台普通的 PC 就行，所需的软件可以从网上下载，免费的，而且也不用注册，安装后马上就能用，而且相当好用。比如作者使用的 Icarus Verilog (Iverilog) 和 Altera 公司的 Quartus II Web Edition 就是这样的软件。如果想要用 FPGA 实现，买些 FPGA 实验板，即可实现诸如"计算机原理与设计实验"等课程所要求的内容。Altera 公司也与国内很多高校合作建立了硬件实验室，免费提供 FPGA 实验板。其实买也不贵。如果使用 Xilinx FPGA 板，则部分代码需要修改。

"如果想自学，读这本书合适吗?" 自学很重要，不管是出于工作需要还是个人兴趣。对已经参加工作的技术人员或在校的非计算机专业的学生来讲，如果想自学，可以慢慢读、慢慢做，因为咱没有期末考试。本书的写作风格就是试图把复杂

的概念通俗易懂地讲出来，让人容易理解，所以本书非常适合自学者阅读。

"如果只想学原理不想学设计，读这本书合适吗?" 对每个基本概念，本书都是先讲它的原理，再讲设计。如果只想学原理，只要不看设计部分就行了。不过还是都学为好，因为设计的过程也是学习的过程。如果读者已经懂了原理，直接看设计部分就行了，这样效率会更高。

"每章最后的习题有没有标准答案?" 没有。有些题目尤其是设计类型的题目的做法有很多种，只要到了罗马 (到了北京也行)，都得 100 分，都是标准答案。有新意的，加 10 分。有些题目附加了提示，仅供读者参考。作者鼓励大家发挥自己的创造性，找一些书上没有的题目来做。

作者欣喜地看到，计算机的硬件实验教学终于开始受到重视，从 2009 年开始在南京大学召开的全国计算机组成与结构课程群实验教学研讨会就是一个证明。作者有幸受邀和参加会议的各位老师进行了交流。大家一致认为，计算机偏硬件类课程在整个计算机学科的教学中居于非常重要的位置，应该从现在起加强对这方面的投入，包括教师队伍的建设、教材的更新以及实验环境的准备等。本书的出版，也算是响应这个会议的号召吧。

希望读者在学完本书之后，在对计算机原理与设计和 Verilog HDL 的理解和使用上，比没有阅读之前有一定程度的提高。从个人的角度讲，熟练地掌握了 CPU 设计技术，至少会增加一种就业机会，而且一不小心就成了专家。从大局着眼，相信读者会为我国的 CPU 设计水平的提高做出自己的贡献。另外，本教科书的这种写法是一种新的尝试，因此作者也期待有更多的与本书风格类似的教科书出现，大家一起来繁荣我国的计算机基础教育事业 —— 也就是清华大学教授李三立院士在"序言"中提到的 —— 把金字塔的基础打牢。

本书得以出版，作者感谢李三立教授。李教授为本书的写作和出版提供了很好的建议和帮助。另外，大约从十年前开始，李教授就为作者联系了很多高校，在暑假期间开设 CPU 设计系列讲座，比如清华大学、东南大学和浙江大学等。作者也感谢清华大学出版社对本书的支持。

本书中所有的 Verilog HDL 源代码均经过 Quartus II Web Edition 或者 Iverilog 的验证，C 语言程序由 gcc 编译。本书初稿在 Linux / CentOS 环境下完成：中文输入用 emacs + chinput，除了从屏幕截取的仿真波形图，其他所有的图几乎都是用 tgif 或 gnuplot 画成，PDF 文件用 VTeX / Linux + CJK 生成。以上均为免费软件，作者一并向这些软件开发者和相关公司表示感谢。

由于作者水平有限，本书所载的 Verilog HDL 源代码写得不太漂亮，而且只是一家之言。希望读者能写出质量更高的代码并对本书的错误之处加以指正。来信请寄：yamin@ieee.org 或者 yamin@computer.org。

李 亚 民

2011 年 5 月 5 日

目　录

第 1 章　计算机基础知识及性能评价

欢迎阅读这本《计算机原理与设计 —— Verilog HDL 版》教科书。本书将从最基本的逻辑电路及其 Verilog HDL 设计讲起，由浅入深，逐步讲到如何用 Verilog HDL 设计一个带有分开的指令及数据 TLB/Cache 和能执行浮点加、减、乘、除及开方指令的浮点部件 FPU 的多核 CPU。本书也给出常用的输入/输出接口的设计方法，包括异步通信接口 UART、PS/2 键盘与鼠标接口、视频图像阵列 VGA 接口、I2C 和 PCI 总线接口。以上 CPU 和 I/O 接口基本上都有 Verilog HDL 源代码，供读者在设计自己的电路时参考。另外，我们也将描述高性能计算机及其互联网络的设计方法。

本章主要讲解一些基本的概念，对计算机原理与设计和计算机性能评价做一个简单的介绍，以便为学习以后各章打下一点基础并留下一个整体印象。

1.1　计算机系统概述

假设你有一台 PC，存储器有 16GB，CPU 主频为 3GHz。问：上述的两个 G 是否相同？本节最后将给出这个问题的答案。在此之前，我们简要介绍计算机和计算机系统各自包含的内容、计算机发展简史、计算机的指令结构以及 RISC 和 CISC 的基本概念与不同之处。

1.1.1　计算机系统的组成

读者可能知道什么是单片机，或者至少听说过 (没听说过也没关系)。单片机的全称是单片计算机 (Single Chip Computer)，它就是一个芯片，内部集成有 CPU (Central Processing Unit)、存储器 (Memory) 和一些常用的 I/O 接口 (Input/Output Interface)。由此可见，包含了以上三种部件的芯片或电路板就叫计算机。

那么，一般的用户能不能直接使用这个计算机呢？答案是不能。我们常说的计算机实际上是指计算机系统 (Computer Systems)。计算机系统中除了包含有计算机 (CPU、存储器和 I/O 接口) 之外，还有计算机软件 (Software) 和 I/O 设备 (Input/Output Devices)。当然还少不了电源。

最基本的软件是操作系统 (Operating System)。它负责管理计算机系统的所有硬件资源，为用户使用计算机系统提供一个界面。最基本的也是最常用的操作系统有 MS-DOS 和 Linux 核 (Kernel)。微软的各种版本的 Windows 或诸如 Fedora 和 CentOS 等可被认为是在基本的操作系统基础上开发的高级图形界面。

软件是用计算机语言 (高级语言或汇编语言) 编写的程序再经过编译或汇编后生成的可执行的二进制代码。编译器 (Compiler) 和汇编器 (Assembler) 本身也是软件，有时也称为软件工具。其他的软件工具还有编辑器 (Editor) 和软件调试器 (Debugger)。计算机系统还配有常用的实用程序库 (Utilities)。除此之外的其他软件可以统称为应用程序 (Applications)。提问：第一个汇编器是用汇编语言编写的吗？

软件的开发过程大致如下。首先选用一种 (或几种) 高级语言 (或汇编语言)，用一个编辑器来编写源程序。编好后存入硬盘。再用编译器或汇编器把它转换成目标码 (CPU 可执行的二进制的机器码)。生成好的机器码程序仍存放在硬盘上。然后通过操作系统执行它，看看软件能否正常工作。第一次执行一般都通不过。这时可用调试器来跟踪程序的执行，找出错误所在。一个成熟的软件要经过成百上千次调试才能趋于完善 (注意是“趋于”)。

I/O 设备 (或称外部设备) 就像一辆车的轮子和方向盘。大家知道，常用的 I/O 设备有键盘、鼠标、显示器、硬盘、打印机。网络接口也可归于此类。不太常用的 I/O 设备有扫描仪和磁带机。由早期不太常用变为常用或越来越常用的 I/O 设备有视频摄像头、扬声器、话筒、CD / DVD 驱动器和优盘 (USB Memory)。实际上，高档手机就是一个微型的计算机系统。

综合以上的描述，我们给出图 1.1 所示的计算机系统的组成结构。有关 CPU 内部的 ALU、FPU、Cache、TLB、寄存器堆和控制部件等内容，将在以后各章详细描述并给出电路设计的 Verilog HDL 源代码。

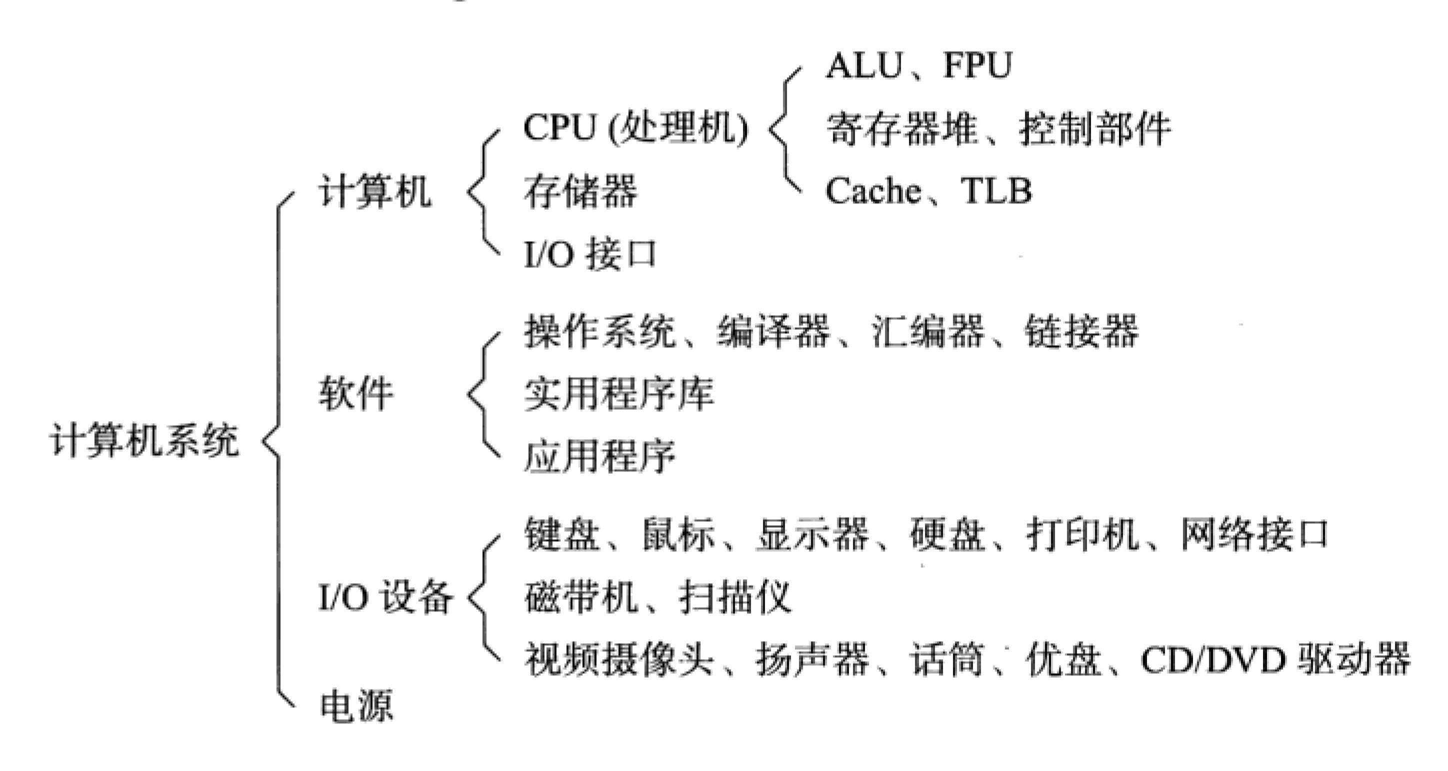


图 1.1 计算机系统组成

好像可以把计算机和 I/O 设备统称为硬件 (Hardware)。这样就可以讲一个计算机系统由硬件和软件组成 (电源不提也罢)。从这个意义上讲，一个大学有了计算机学院再成立一个软件学院也是有道理的。不过两个学院要紧密协作，因为计算机系统的设计要讲究 Hardware/Software Co-Design。外设学院好像还没听说过。

1.1.2 计算机发展简史

计算机经历了机械时代和机电时代，于 20 世纪 40 年代进入电子时代 (第一代)，其标志是 40 年代中期使用电子管的计算机 ENIAC 的诞生。第一代计算机没有编程工具，程序员必须通过控制面板上的开关来向计算机输入指令。

20 世纪 50 年代后期，计算机进入第二代。其主要标志是使用分立的晶体管和磁芯存储器，典型的计算机是 IBM 7000 系列和 CDC 6000 系列。操作系统和高级语言在这期间诞生。晶体管的发明也是一件具有里程碑意义的事情。

20 世纪 60 年代中期，计算机进入第三代。其主要标志是使用了集成电路 (Integrated Circuit)，即把许多晶体管集成在一个芯片内。有名的生产集成电路的公司有仙童 (Fairchild) 和德州仪器 (Texas Instruments)。典型的计算机是 CDC 的 6600 和 IBM 360 (后来变成了 370)。IBM 在 360 上首次把计算机体系结构 (Computer Architecture) 和计算机组成 (Computer Organization) 分开，不同的型号具有相同的体系结构，售价从几十万到上百万美元不等。

给个具体的价格以加深印象，感觉一下你现在有多幸福：IBM System/360 Model 75 的时钟频率是 5.1MHz，存储器为 256KB～1MB，售价是当时的 190 万美元，而性能远不及现在普通的 PC (Personal Computers)。如果你有本事带一台 PC 以超过光速的速度通过时间隧道回到 20 世纪 60 年代去卖，你就变成富人了。从另一个角度看，说不定现在也有一百年以后的人在向我们兜售计算机呢。大家想想，那一百年以后的计算机会是什么样的呢？

20 世纪 70 年代末期，计算机进入第四代。其主要标志是使用了超大规模集成 (Very Large Scale Integration，VLSI) 电路和半导体存储器。典型的计算机是使用微处理机的低端个人计算机和高端的 IBM 390 系列。IBM PC 使用了Intel 的 8086 微处理机，从此 Intel x86 指令结构开始流行。

虽然还是 IBM 公司，但这次是为他人做了嫁衣。IBM PC 使用了 Intel 的 8086 CPU 和微软的操作系统 MS-DOS，而且随机赠送电路图。这样就成全了 Intel 和微软以及其他生产 PC 的公司。IBM 好像没得到什么好处，也许它认为 PC 只是一个小玩具，要把精力放在做大做强上。如果 IBM 当初使用自己的 CPU 和操作系统做 PC，也公开电路图，那么今天就会有另外一种景象。当时与 Intel 8086 同样有名的微处理机还有 Zilog 的 Z80 和 Motorolar 的 6800，不过它们没有 8086 的运气好。从现在高性能 CPU 设计的角度看，x86 指令系统不是很好甚至很不好。不好的原因稍后讲。

第五代计算机从什么时候开始、以什么为标志，以及现在是第几代等，众说纷纭。作者也不在这里多说了。总之，现在的计算机，不管是高档 PC 还是超级计算机，都使用多核多线程的 CPU。还是那句话，以后的计算机又会是什么样的呢？希望寄托在读者身上。

这里有必要简单地提一下摩尔定律 (Moore’s Law)。摩尔是 Intel 公司的创始人之一，他在 1965 年提出了半导体芯片上所能集成的晶体管的数量每两年翻倍的预测。经修正后变为每 18 个月翻一番。后来的实际情况证实了这个预测是正确的。不仅是晶体管数量，超级计算机的性能也大致是每 18 个月翻一番，而且每隔 18 个月相同的钱所能买到的性能也翻一番。

如果你现在想买一台 PC，你可能会想，明年买会更便宜，或相同的钱能买到性能更好的。有这样的想法很自然，但如此，你一辈子也买不上 PC，这是因为明年还有明年。最佳答案是今天就买、回家就用。

1.1.3 计算机指令结构

软件的执行是 CPU 的工作。CPU 能执行的实际上是它能理解的指令 (Instruction)。不同的 CPU 有不同的指令系统，它们的格式以及用二进制位的表示也是不同的。一般地，一条指令都有一个指令操作码 (指定该指令要完成什么样的操作)，其余部分可以是立即数 (Immediate)、寄存器号、存储器地址或者它们的组合，即给出如何得到操作数以及把指令的操作结果存到什么地方等信息。

立即数是直接参加运算的数据，在指令中给出。立即数的二进制位数从 8 到 32 位不等。CPU 中有若干个寄存器，它们能够保存数据，并且速度非常快。寄存器的数量依 CPU 不同而有所差别，一般从 8 到 256 个不等。立即数是常数 (一旦定了就不能改了)，而寄存器操作数是变量。存储器的地址一般是 32 位，有些 CPU 直接在指令中给出，有些是通过寄存器的内容与立即数相加得到。存储器数据也是变量，但把常数放在存储器中也未尝不可。

指令的种类大致有：(1) 整数算术运算和逻辑运算；(2) 寄存器与存储器之间的数据传送；(3) 条件转移和无条件跳转；(4) 子程序调用和返回；(5) 浮点运算；(6) I/O 访问；(7) 系统维护，比如系统调用和从中断返回等指令。

我们通过 x86 和 MIPS 指令系统的具体的例子加以说明。下面是用 C 语言编写的一个乘法子程序，实现两个 16 位无符号数相乘，结果为 32 位的无符号数。

```
unsigned int mul16 (unsigned int x, unsigned int y) {
    unsigned int a, b, c;
    unsigned int i; // counter
    a = x; // multiplicand
    b = y; // multiplier
    c = 0; // product
    for (i = 0; i < 16; i++) { // for 16 bits
        if ((b & 1) == 1) { // LSB of b is 1
            c += a; // c = c + a
        }
        a = a << 1; // shift a 1-bit left
        b = b >> 1; // shift b 1-bit right
    }
    return(c);       // return product
}
```

当然，你可以直接用 c = a * b 实现相乘，但我们这里演示的是如何用加法和移位操作实现相乘。如果我们的计算机系统中的 CPU 具有 x86 指令结构，经过 gcc −O4 −S 的编译 (GNU C 编译器，优化级别 4) 后，生成如下的 x86 汇编语言程序。

```
mul16:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %ecx
```

```
        pushl   %ebx
        movl    12(%ebp), %edx
        xorl    %ebx, %ebx
        movl    $15, %eax
        .p2align 2,,3
.L6:
        testb   $1, %dl
        je      .L5
        addl    %ecx, %ebx
.L5:
        sall    %ecx
        shrl    %edx
        decl    %eax
        jns     .L6
        movl    %ebx, %eax
        popl    %ebx
        leave
        ret
```

我们并不解释所有的指令，只把一条 addl %ecx, %ebx 指令拿出来晒晒。这条指令实现 C 语言程序中的 c += a，其中 addl 指出是加法指令，%ecx 和 %ebx 是两个 32 位源操作数寄存器，%ebx 也是目的寄存器。该指令把 %ecx 和 %ebx 寄存器中的两个 32 位数据相加，把结果存入 %ebx 寄存器。由此我们看出，x86 的指令具有双操作数的结构。如果把相同的 C 语言程序用 gcc –O4 –S 命令编译成 MIPS 汇编程序，我们将得到以下的代码。

```
mul16:
        .frame  $sp, 0, $31    # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000, 0
        .fmask  0x00000000, 0
        .set    noreorder
        .set    nomacro
        move    $6, $0
        li      $3, 15         # 0xf
$L6:
        andi    $2, $5, 0x1
        addiu   $3, $3, -1
        beq     $2, $0, $L5
        srl     $5, $5, 1
        addu    $6, $6, $4
$L5:
        bgez    $3, $L6
        sll     $4, $4, 1
        j       $31
        move    $2, $6
```

实现 C 语言程序中 c += a 语句的指令是 addu $6, $6, $4。该指令的一般格式是 addu rd, rs, rt，其中 rd 是目的寄存器，rs 和 rt 是两个源操作数寄存器。由此可以看出，MIPS 的指令具有三操作数的结构。几乎所有的 RISC 类型的 CPU 都具有三操作数的指令结构。注意以上代码中指令的次序：处在转移或跳转指令 beq、bgez 和 j 下面的指令 srl、sll 和 move 在转移或跳转发生之前执行。这是为了提高流水线效率的所谓的延迟转移 (Delayed Branch)。汇编程序中 0x 是十六进制的意思。

还有一操作数的指令格式。在这种 CPU 中，有一个称为累加器 (Accumulator) 的寄存器，它总是参加运算并保存结果，是第一把手且不用参加选举而永远当选的，因此只要在指令中指定另外一个操作数就够了。6502 和 Z80 CPU 就属于此类。

有没有零操作数的指令格式呢？有。在这种 CPU 中，有一个称为堆栈 (Stack) 的快速存储器。堆栈有所谓栈顶，栈顶的位置由栈顶指针指出。一条运算指令只要指明做何种运算就行，不用指定任何操作数。运算时，从栈顶弹出两个数据，运算的结果再压入栈顶。栈顶指针根据运算指令自动调整。Java 虚拟机 JVM (Java Virtual Machine) 的指令系统 Bytecode 就属于此类。

表 1.1 以 add 指令为例总结了上述四种指令的汇编格式，其中的 x，y，z 既可以是存储器地址也可以是寄存器编号。

表 1.1 以操作数个数对指令进行分类

面向通用寄存器或存储器		面向累加器	面向堆栈
三操作数指令	二操作数指令	一操作数指令	零操作数指令
add x, y, z	add x, y	add x	add

所有的指令均用二进制表示。在 x86 中，32 位的通用寄存器只有 8 个 (不是严格意义上的通用)，它们分别是 eax，ebx，ecx，edx，ebp，esp，esi 和 edi。因此寄存器的编码只需要 3 位 ($\log_2 8 = 3$ 或 $2^3 = 8$)，而且常用的指令尽量使用较短的操作码。以下是上述 x86 汇编程序中每条指令的二进制表示。我们可以看出，x86 指令的长度 (二进制位数) 不是固定的，有些指令只用 8 位表示，但有些指令需要几十位。如果不对当前指令译码，就不会知道下一条指令的起始地址。这将给 CPU 的设计，尤其是流水线 CPU 的设计带来不小的麻烦。

```
mul16:
        pushl   %ebp            ; 01010101
        movl    %esp, %ebp      ; 1000100111100101
        movl    8(%ebp), %ecx   ; 100010000100110100001000
        pushl   %ebx            ; 01010011
        movl    12(%ebp), %edx  ; 100010110101010100001100
        xorl    %ebx, %ebx      ; 0011000111011011
        movl    $15, %eax       ; 10111000
        .p2align 2,,3           ; 00001111000000000000000000000000
                                ; 100011010111011000000000
.L6:
```

```
        testb   $1, %dl          ; 1111011011000010000000001
        je      .L5              ; 0111010000000010
        addl    %ecx, %ebx       ; 0000000111001011
.L5:
        sall    %ecx             ; 1101000111100001
        shrl    %edx             ; 1101000111101010
        decl    %eax             ; 01001000
        jns     .L6              ; 0111100111110010
        movl    %ebx, %eax       ; 1000100111011000
        popl    %ebx             ; 01011011
        leave                    ; 11001001
        ret                      ; 11000011
```

以下是 MIPS 汇编程序中每条指令的二进制表示。MIPS 有 32 个寄存器，寄存器号在指令中用 5 位表示。我们可以看出，MIPS 指令的长度是固定的，每条指令都是 32 位。MIPS 的指令格式非常规整，这有利于流水线 CPU 的设计。我们可以讲，设计一个流水线的 MIPS CPU 要比设计一个流水线的 x86 CPU 简单得多。这也正是本书选择 MIPS 指令来讲述计算机原理与设计的原因之一。

```
        move    $6, $0       # 00000000000000000011000000100001
        li      $3, 15       # 00100100000000110000000000001111
$L6:
        andi    $2, $5, 0x1  # 00110000101000100000000000000001
        addiu   $3, $3, -1   # 00100100011000111111111111111111
        beq     $2, $0, $L5  # 00010000010000000000000000000010
        srl     $5, $5, 1    # 00000000000001010010100001000010
        addu    $6, $6, $4   # 00000000110001000011000000100001
$L5:
        bgez    $3, $L6      # 00000100011000011111111111111010
        sll     $4, $4, 1    # 00000000000001000010000001000000
        j       $31          # 00000011111000000000000000001000
        move    $2, $6       # 00000000110000000001000000100001
```

虽然 MIPS 指令比有些 x86 指令需要更多的二进制位来表示，但每条 MIPS 指令的功能非常简单。我们称 MIPS 属于 RISC 类型，而 x86 属于 CISC 类型。

1.1.4 CISC 和 RISC

CISC (Complex Instruction Set Computer) 是对那些具有复杂指令系统的 CPU 的总称。指令系统复杂包含两层意思。一是指一条指令能完成复杂的操作，例如从存储器取来数据、计算、再把结果存入存储器。另外一层意思是指令的格式不规范，操作码及整个指令的长度不统一。CISC 的例子有 Intel x86 (IA-32 和 IA-64)、Motorola MC68000、PDP-11 和 VAX，后两个到 1980 年代已不再生产。

CISC 指令系统试图提供与某些高级语言的语句相对应的指令，使一条指令能完成尽可能多的操作。比如一条指令可以实现计数器减 1，如果计数值不为 0 则跳

转。再比如为了提高对数组元素计算的效率，一条指令可以实现多个操作：使用一个寄存器与立即数相加得到存储器地址，再去访问存储器，把取来的数据与另外一个寄存器的内容相加，并且修改第一个寄存器的内容 (如果是字节操作，则加 1 或减 1；如果是半字操作，加 2 或减 2；如果是字操作，加 4 或减 4)，指向下一个数组元素。CISC 指令系统往往提供丰富的寻址方式 (Addressing Modes)。

当时的存储器的容量很小而且价格昂贵，所以指令系统的设计者想方设法要使编译或汇编过后的程序变短，以使相同容量的存储器能存放更多的指令。这就导致了每条指令长短不一：常用的指令较短，不常用的指令较长。

由于 CISC 的指令过于复杂，设计 CPU 时往往采用微程序 (Microprogram) 的方法来实现这些指令。一条指令被分成若干条微指令 (Microcode 或 Microinstruction)，使用微程序计数器来读出微指令。这样，一个周期就只能执行一条指令的若干分之一。依指令复杂程度的不同，实现一条指令的微指令的数量也不同。微程序的方法非常不适合流水线操作。在流水线 CPU 中，一个周期同时执行多条指令，每条指令处在不同的执行阶段。因此微程序方法只被用在非流水线 CPU 的设计中。CISC 在当时的优点是代码紧凑，使用较少的存储器。缺点是实现复杂的指令需要较多的芯片面积而且不利于流水线操作。

是不是 CISC 所有的指令都是复杂的指令呢？也不是，简单的指令也不少。一般地讲，实现 CISC 简单指令所需的电路占总体的 20%，而其余的 80% 用来实现 CISC 复杂指令。那么编译器能否有效地使用这些复杂的指令呢？答案是不能。原因是指令太复杂了，不知怎么用。也是一般地讲，经过编译后的程序 80% 都是简单的指令，20% 才是复杂的指令。这就是所谓的 82-28 现象。我们可以看出，在 1.1.3 小节的 x86 汇编程序的例子中，几乎所有的指令都是简单的指令。

RISC (Reduced Instruction Set Computer) 是对那些具有简单指令系统的 CPU 的总称。该名称是 UC Berkeley 的 David Patterson 首先提出的。而 CISC 是有了 RISC 的名称后派生出来的名称，人家当初并没有称自己的 CPU 是 CISC。在 20 世纪 80 年代初，Patterson 领导的课题组对当时的 CPU 的指令系统做了研究，提出了 RISC 概念，并试做了 RISC-I 和 RISC-II CPU。Sun Microsystems 的 SPARC CPU 的指令系统就是在 RISC-I/II 的基础上设计的。与此同时，Stanford 的 John Hennessy 也做着类似的研究，提出了 MIPS (Microprocessor without Interlocked Pipeline Stages) 指令系统并一度离开 Stanford 创建了 MIPS 公司。后来两个人合写了两本著名的教科书[3, 27]。实际上，早在 1975 年，IBM 的 John Cocke 领导的小组就开始了对基于简单指令系统的微处理机体系结构的研究，其项目名称是 IBM 801。因此有人，尤其是 IBM 的人，称 John Cocke 为“RISC 之父”。

RISC 指令系统的特点有两个。一是指令长度固定；二是所谓的 Load/Store 结构。Load/Store 结构是指一条 Load 指令把存储器的数据取到 CPU 的寄存器中，一条 Store 指令把寄存器的数据存到存储器中。这两条指令都不对数据本身做任何运算，相当于有诚信的快递公司为客户运送物品。而运算指令的操作数都在寄存器中。指令长度固定有利于流水线 CPU 的设计。自从 RISC 问世之后，微程序就很少被使用

了。20 世纪 80 年代后半段 RISC CPU 陆续登场，有 MIPS R2000/R3000、HP PA-RISC 和 Sun Microsystems SPARC。20 世纪 90 年代 64 位 RISC CPU 登场，有 IBM/Motorola PowerPC、MIPS R4000/R8000、Sun Microsystems SuperSPACR/UltraSPARC II、DEC Alpha 和 HP PA-RISC7200 等。

那么，是否 RISC 已经取代了 CISC 呢？没有。主要原因是市场。CISC，尤其是 x86，有大量的软件资产可以继承，而 RISC 没有。想象一下将 x86 指令系统彻底抛弃会带来什么后果。实际上，进入 21 世纪后生产的 CPU，比如 UltraSPARC III、IBM POWER4/POWER5/POWER6、Intel Pentium 4/Xeon/Itanium/Core 2/Core i3/i5/i7/i9 和 AMD Athlon/Opteron，都有一个趋势，就是 RISC 越来越 CISC、CISC 越来越 RISC 了。意思是 RISC CPU 加了大量的复杂的指令，而 CISC CPU 并不直接执行 CISC 指令，而是转换成 RISC 指令再执行，比如把 x86 指令转换成 μop。

有意思的是由 6502 (CISC) 派生出来的 ARM 具有复杂的指令结构，但人们喜欢把它归类到 RISC。ARM 在嵌入式市场有压倒性的占有率。有关早期的 RISC CPU 的详细描述，请参阅《RISC —— 单发射与多发射体系结构》[35]。

1.1.5 一些基本单位的意义

现在回答本节开始处提出的问题。答案是两个 G 不一样：3GHz (CPU 主频) 中的 $G = 10^9 = 1\,000\,000\,000$，而 16GB (存储器容量) 中的 $G = 2^{30} = 1\,073\,741\,824$，比 10^9 大一些。当表示时钟频率或时钟周期时，用 10 的多少次方；当表示存储器的容量时，用 2 的多少次方。表 1.2 列出了一些基本单位的意义。

表 1.2 一些基本单位的意义

存储器容量				时钟频率			周期长度		
K	kilo	2^{10}	1 024	K	kilo	10^3	m	milli	10^{-3}
M	mega	2^{20}	1 048 576	M	mega	10^6	μ	micro	10^{-6}
G	giga	2^{30}	1 073 741 824	G	giga	10^9	n	nano	10^{-9}
T	tera	2^{40}	1 099 511 627 776	T	tera	10^{12}	p	pico	10^{-12}
P	peta	2^{50}	1 125 899 906 842 624	P	peta	10^{15}	f	femto	10^{-15}
E	exa	2^{60}	1 152 921 504 606 846 976	E	exa	10^{18}	a	atto	10^{-18}
Z	zetta	2^{70}	1 180 591 620 717 411 303 424	Z	zetta	10^{21}	z	zepto	10^{-21}
Y	yotta	2^{80}	1 208 925 819 614 629 174 706 176	Y	yotta	10^{24}	y	yocto	10^{-24}

当我们说到存储器的容量时，总是以字节 (Byte，8 位) 为单位。另外还有半字 (Half Word)，是 16 位；字 (Word)，32 位；长字 (Long Word) 是 64 位。但 x86 定义长字是 32 位，而字为 16 位。这都是 8086 惹的祸。16G 字节可以写为 16GB。二进制位的英文单词是 Bit，它是一个由 Binary digit 造出来的词，即一个“二进制数字”的意思。有人把它翻译成“比特”，搞得神神秘秘，专门用来蒙人。

1.2 计算机的基本结构

我们已经讲过，计算机中包含有 CPU、存储器和 I/O 接口。本节简要介绍这三大件。顺便提一下，CPU 是一个比较老的名称，自从处理机 (Processors) —— 尤其是微处理机 (Microprocessors) —— 的名称问世之后，就很少有人再使用 CPU 这个名称了，用了连自己也觉得挺土的。好像 CPU 和处理机各领了风骚三十年。但最近 CPU 的名称又开始时髦起来了，真是应了那句话：三十年河东，三十年河西。CPU 和处理机好像有些微妙的差别，比如早期的 CPU 不是一个芯片，而是一块电路板，由很多分立元件组成 CPU，而处理机是一个芯片。但现在 CPU 也是指芯片，而且都集成有 Cache (一种小容量高速度的存储器) 和 TLB (一种用于地址转换的缓冲区)。所以也别管那么多了，认为二者指的就是那个能执行指令的芯片就行了。

1.2.1 RISC CPU 的基本结构

CPU 负责执行指令。图 1.2 是简化的 RISC CPU 的结构图。

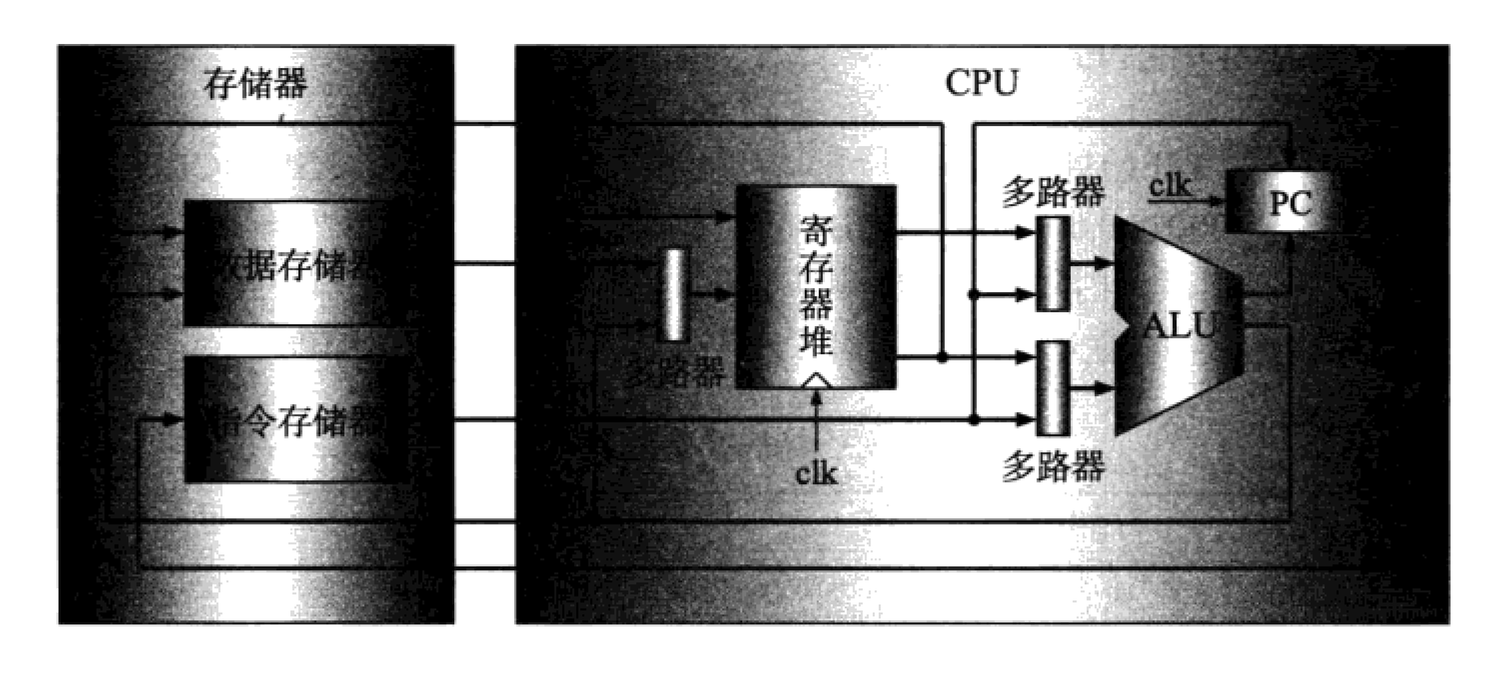


图 1.2 简化的 RISC CPU 结构图

指令从指令存储器取来，存储器的地址由程序计数器 PC (Program Counter) 提供。小三角形处接时钟信号。寄存器堆 (Register File) 里有一堆寄存器，存的是数据。ALU (Arithmetic Logic Unit) 负责计算，两个数据可以全部来自寄存器堆，其中的一个也可以来自指令中的立即数。图中的多路器 (Multiplexer) 从多个输入中选择一个送出。计算的结果存入寄存器堆。如果是 Load 指令，则 ALU 的输出作为地址使用，访问数据存储器，把从数据存储器取来的数据存入寄存器堆。如果是 Store 指令，把寄存器的内容存入数据存储器。Load 和 Store 指令在计算存储器地址时把一个寄存器的内容与立即数相加，因此寄存器堆的第二个输出可以用来把寄存器数据送到数据存储器。如果是条件转移指令，则要根据 ALU 输出的标志位来判断是否转移，转移地址由当前 PC 和带符号的立即数 (或称偏移量) 相加得到。不转移时 PC 要

加 4，指向下一条指令 (一条 32 位的指令占用 4 个字节)。图 1.2 中的电路实际上是单周期 CPU，即一个时钟周期执行一条指令，一条指令执行完再执行下一条指令。

单周期 CPU 是在一条指令的所有操作全部完成后，才开始下一条指令的执行。流水线 CPU 是把一条指令的执行过程分成若干阶段，或称级 (Stage)，使得多条指令能重叠执行。图 1.3 示出的是一个简化的流水线 RISC CPU 的结构。

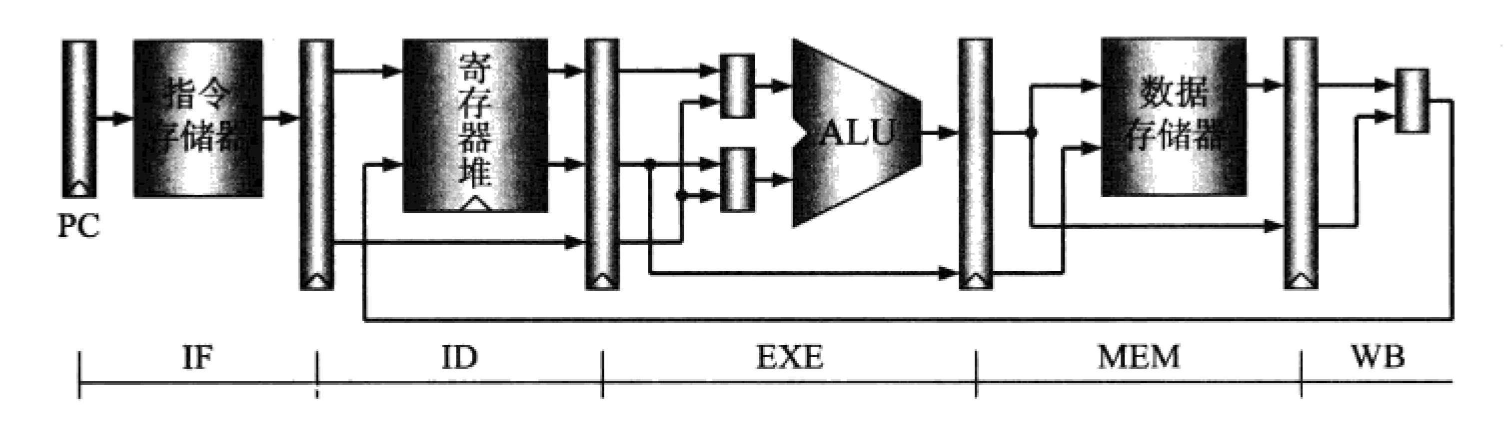


图 1.3 简化的流水线 RISC CPU 结构图

图中的流水线有 5 级，它们分别是：(1) 取指令 IF (Instruction Fetch) 级；(2) 指令译码 ID (Instruction Decode) 级；(3) 执行 EXE (Execution) 级；(4) 存储器访问 MEM (Memory Access) 级；(5) 结果写回 WB (Write Back) 级。级与级之间设置有流水线寄存器，用来保存中间结果。一级占用一个周期。这样的流水线 CPU 能同时执行 5 条指令，每条指令处在不同的级，从而增大指令的吞吐率。

从 1980 年开始，CPU 的速度提高很快，但动态存储器 DRAM (Dynamic Random Access Memory) 的速度改进不大，这就造成了二者之间不匹配。因此现在的 CPU 芯片中都集成有指令 Cache 和数据 Cache，见图 1.4。Cache 是小容量的高速存储器，用来存放经常使用的指令或数据。如果在 Cache 中存放的指令或数据第二次被用到，则不用访问存储器，直接从 Cache 中得到，从而加快了访问速度。

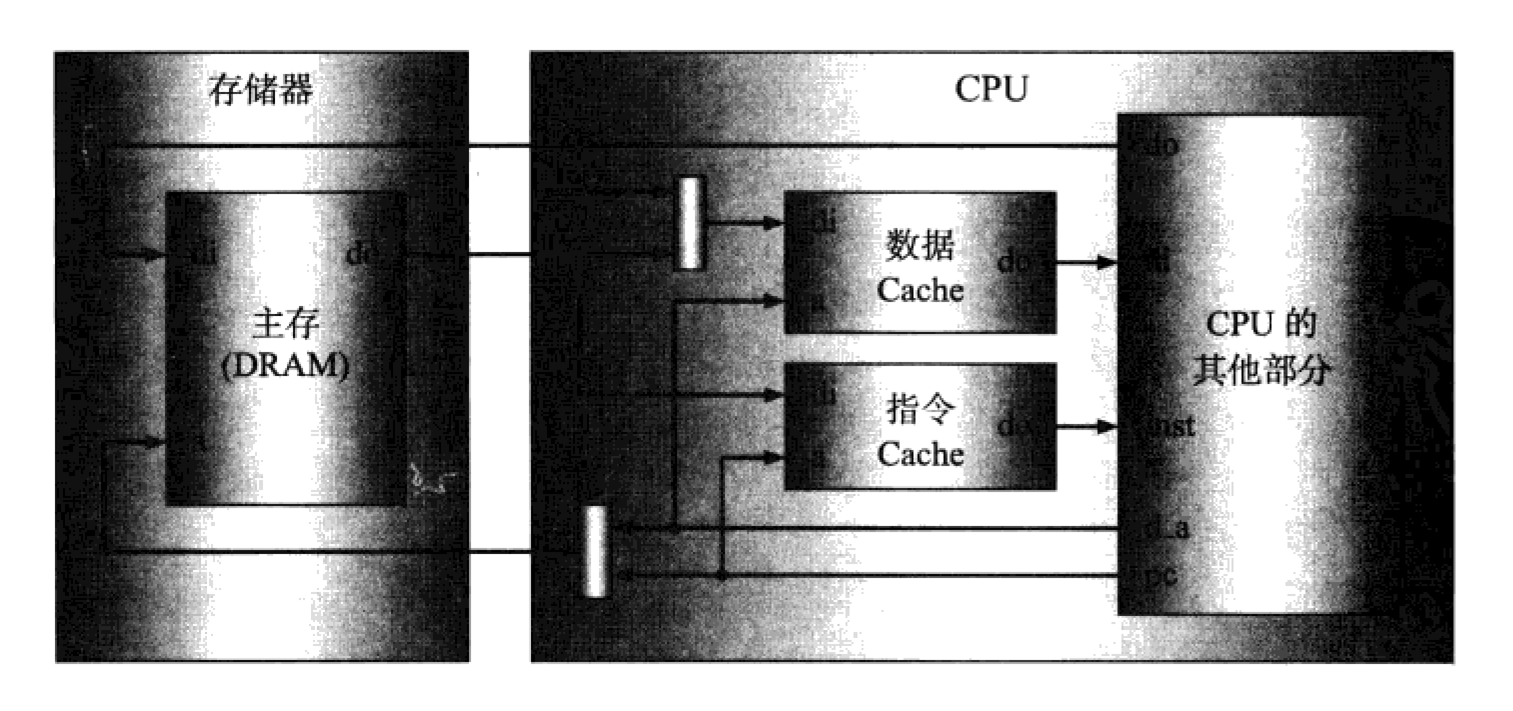


图 1.4 CPU 中的指令 Cache 和数据 Cache

现在的计算机中的主存 (Main Memory) 一般都使用 DRAM。DRAM 具有容量大且价格便宜等优点，缺点是速度慢且控制时序复杂。有钱人会使用静态存储器 SRAM (Static RAM)。SRAM 价格昂贵且费电，但速度快且控制电路简单。

1.2.2 多线程 CPU 和多核 CPU

与单周期 CPU 相比，流水线 CPU 大大地提高了性能，它的理想性能是每个时钟周期执行一条指令。超标量 (Superscalar) CPU 试图在一个周期取出多条指令并行执行。但由于指令之间的相关性，即一条指令使用前一条指令的结果，超标量 CPU 的性能大概是一个周期能执行 1.2 条指令 (统计值)。而为了取得这 20% 的性能改善，超标量 CPU 要增加大量的硬件电路来调度这些同时取出的指令，比如使用寄存器重命名、预约站、重排序缓冲区等。超标量 CPU 不可能再进一步提高性能了，这是由指令级的并行度 ILP (Instruction Level Parallelism) 所决定的。即使编译器可以使用诸如循环展开等优化技术，超标量 CPU 对性能的改善也很有限。

多线程 (Multithreading) CPU 试图并行执行多个线程。一个线程就是一段能够独立执行的程序，再加上执行时所需的环境。一个简化的能够同时执行四个线程的多线程 CPU 的结构在图 1.5 中给出。

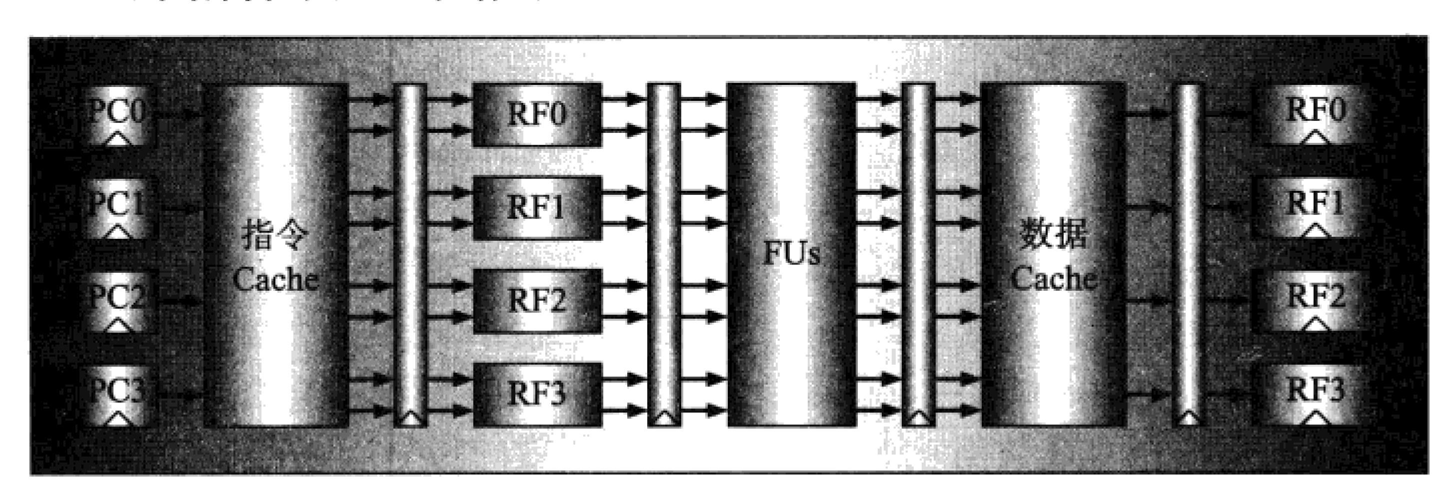


图 1.5 简化的多线程 CPU 结构示意图

图 1.5 中的 PC 是程序计数器，RF 是寄存器堆，FU (Functional Unit) 是功能部件，比如 ALU 和 FPU (Floating Point Unit)。没标名字的是流水线寄存器，用来保存中间运算结果。多线程 CPU 的特点是所有线程共享功能部件和 Cache，这有利于提高这些部件的使用效率，但增加了硬件设计的复杂度。总之，多线程 CPU 试图通过线程级的并行度 TLP (Thread Level Parallelism) 来提高 CPU 的性能。

多核 (Multi-Core) CPU 是在一个芯片中集成多个核 (Core)。一个核就相当于一个普通的 CPU[1]。一个简化的四核 CPU 的结构在图 1.6 中给出，所有的核可以共享第二级 Cache，图中没有画出。

我们可以认为多核 CPU 是把小规模的多处理机 (Multiprocessors) 集成在一个芯片上。学术界最早使用的名称是 Chip-Multiprocessors。与多线程 CPU 相比，多核

[1]普通的 CPU 不是多核 CPU，否则该定义就没完没了地递归了。

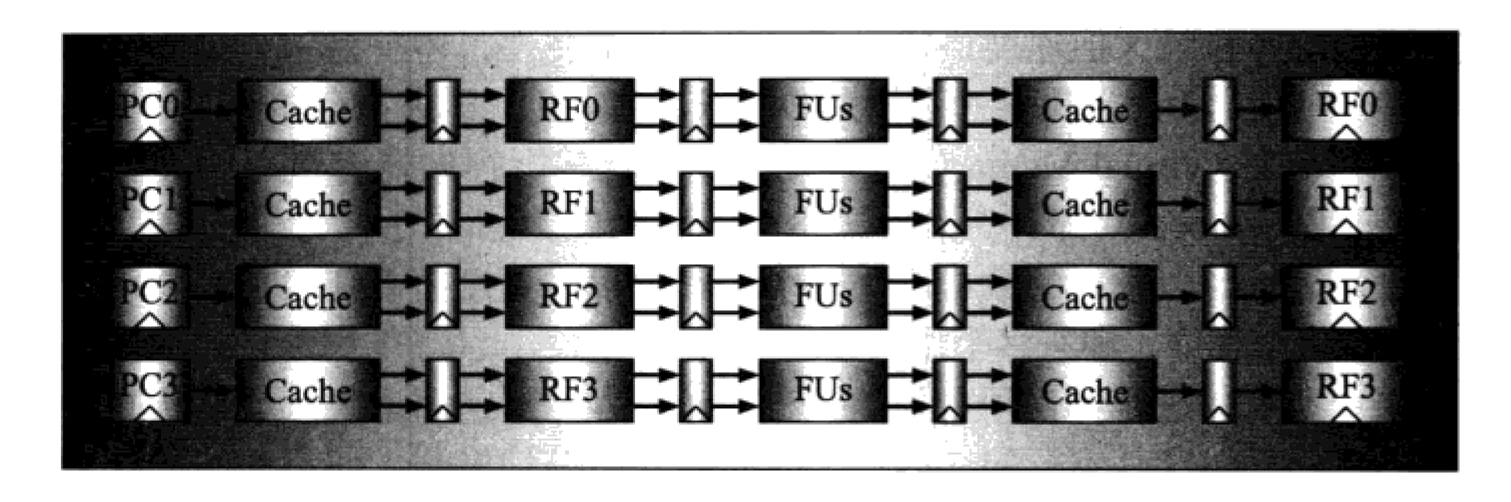


图 1.6 简化的四核 CPU 结构示意图

CPU 具有设计简单的优点，但功能部件的利用效率没有多线程 CPU 高。

我们可以很自然地想到，多核 CPU 中的每个核又可以是一个多线程 CPU。这样的 CPU 可称做多核多线程 CPU，或多线程多核 CPU (名称有些过长)。IBM、Intel 和 AMD 都生产了这样的 CPU。

一般地讲，多核 CPU 中的每个核可以独立工作，因为它本身就是一个 CPU。这样的 CPU 可以实现多指令流多数据流 MIMD (Multiple Instruction Streams on Multiple Data Streams) 功能。我们不妨把诸如 IBM Cell CPU 也称做多核 CPU。Cell CPU 中有多个处理单元 PE (Processing Element)，所有的 PE 执行相同的指令，对多个数据同时做同样的运算。我们称这样的 CPU 实现的是单指令流多数据流 SIMD (Single Instruction Stream on Multiple Data Streams) 功能。

1.2.3 存储层次和虚拟存储器管理

存储器 (主存) 是用来暂时存放正在执行的程序的地方。一个程序包含指令和数据。CPU 从存储器读出指令，对数据进行计算。前面我们已经讲过，由于存储器的速度比 CPU 慢得多，因此 CPU 片内集成有小容量的高速存储器，即 Cache。

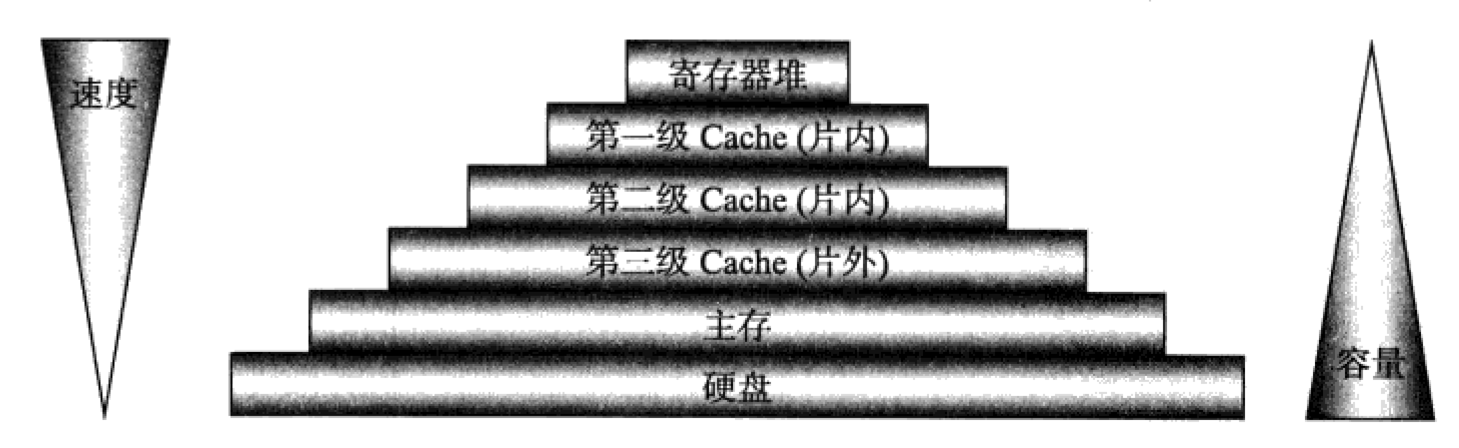


图 1.7 存储层次的概念

图 1.7 示出的是典型的存储层次。Cache 有片内 (On-Chip) 和片外 (Off-Chip) 之分。片是指 CPU 芯片。片内 Cache 可以有两级，第一级一般是分开的指令 Cache 和数据 Cache，第二级 Cache 由指令和数据共享。寄存器堆的速度最快，不过它只存放

数据。硬盘的容量最大但速度最慢。硬盘除了保存文件，还有另一个重要的工作，就是与主存一起，为用户提供一个较大的虚拟存储空间，即虚拟存储器。

虚拟存储器 (Virtual Memory) 允许用户使用比实际存储器容量还要大的存储空间。每个进程[2] (Process) 都有一个从地址 0 开始的虚拟地址空间。我们知道，当一个计算机投入运行时，进程的数量不止一个，其中很多是操作系统的进程。用户也可以同时执行多个程序，即有多个用户进程。

如果令所有的进程都使用虚拟地址直接访问存储器，那么大家就都从 0 地址开始使用主存。显然这是不行的。因此，我们需要有一种机制，使不同的进程使用主存的不同区域。这就需要对进程的虚拟地址进行转换，用转换后的实际地址去访问存储器。

在以“页”(Page) 为单位对存储器进行管理的机制中，虚拟地址空间和主存都被分成大小固定的页。这样，一个虚拟地址就可以分成两部分：高位部分是页号，低位部分是页内的偏移量。对虚拟地址进行转换时，只需转换页号就行了。

操作系统有一个很大的页表 (Page Table)，用来实现地址转换。方法是使用虚拟页号访问页表，从中得到实际页号。那么页表在什么地方呢？也在主存中。有的页表还分成若干层。那么地址转换岂不是要花很长的时间吗？是。因此现在的 CPU 都集成有类似于 Cache 的高速缓冲区，专门用于地址转换。这个缓冲区的名称是 TLB (Translation Lookaside Buffer)，IBM 起的。

图 1.8 是使用 TLB 转换存储器地址的概念图。图中还列出了其他的一些名称，分成左右两组。每组中的名称在一般情况下都有相同的意义，但依 CPU 不同，这些名称的意义可能会有所不同。

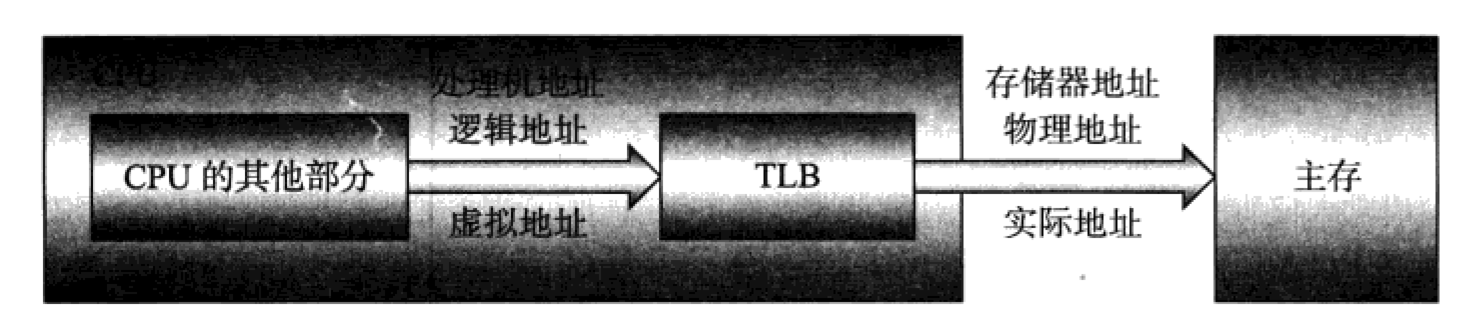


图 1.8 使用 TLB 转换存储器地址

1.2.4 I/O 接口和总线

CPU 通过 I/O 接口来访问 I/O 设备。一个计算机系统中有很多 I/O 设备，因此在系统中往往设置有总线 (Bus)。所有的 I/O 设备通过 I/O 接口连到总线上，如图 1.9 所示。一条总线除了有地址/数据线和读写控制线，还有一些用于同步的信号线。图中画有两级总线，它们之间用总线接口连接。

当 I/O 设备准备好要与 CPU 通信时，可以通过 I/O 接口向 CPU 发出中断请求。CPU 收到请求后，停止当前程序，转去执行中断处理程序。在中断处理程序

[2]进程是操作系统中最基本的概念。一个进程是指一个正在执行的程序和执行时所使用的环境。

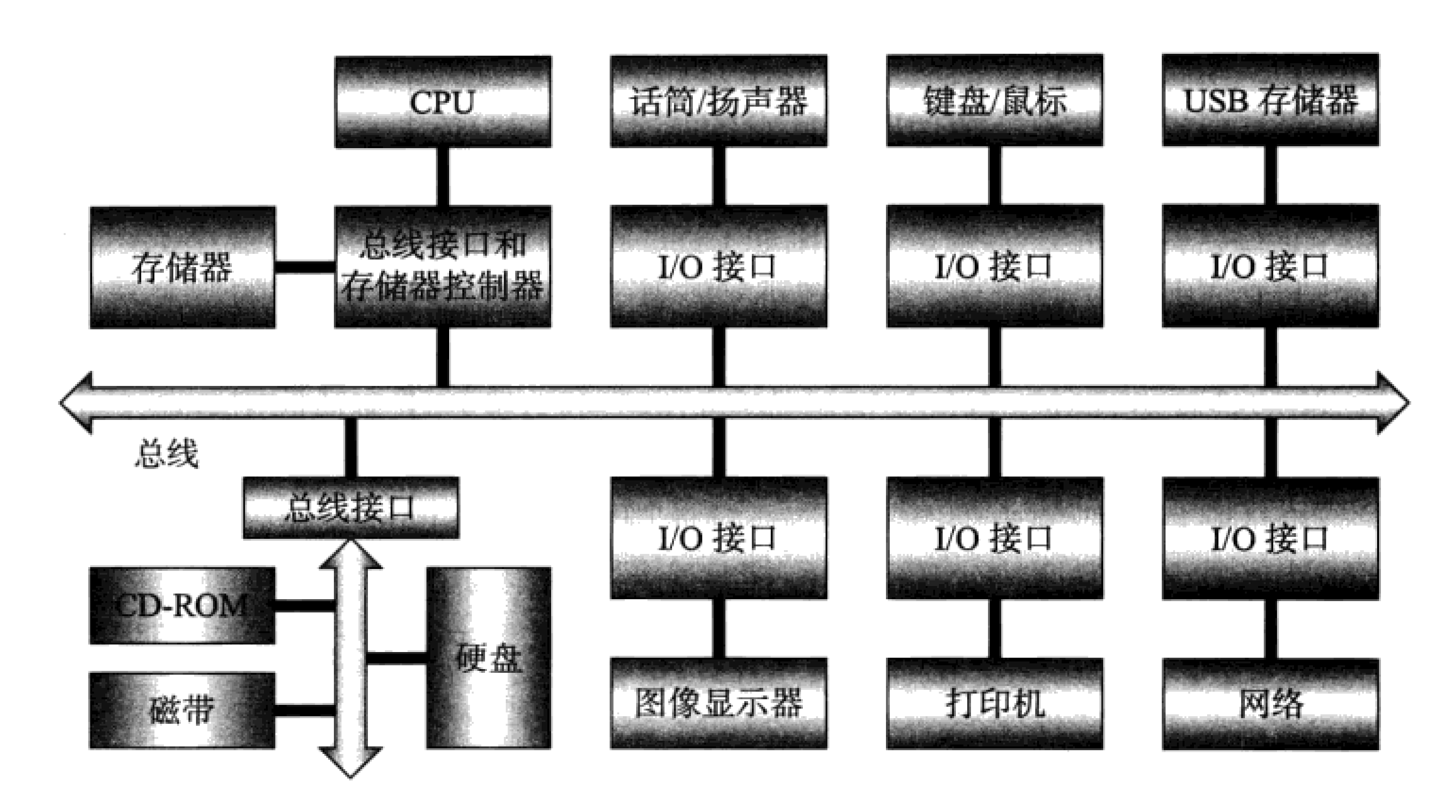


图 1.9 I/O 接口在计算机系统中的位置

中，CPU 可以使用指令来读写 I/O 数据，然后返回到原来的程序继续执行。读写 I/O 数据的指令依 CPU 的不同而不同。在使用存储器映像的 I/O 空间的情况下，CPU 可以用和访问存储器相同的 Load/Store 指令来读写 I/O 数据。如果使用与存储器空间分开的 I/O 空间，则必须要有专门的 I/O 指令，比如 x86 的 in 和 out 指令。

1.3 如何提高计算机的性能

提高计算机的性能可以从两方面着手。一是改进集成电路的工艺，减小线宽，增加晶体管数量以及提高 CPU 的时钟频率。二是改进体系结构，减少一条指令执行所需要的平均的时钟周期的数量，比如使用流水线、超标量、多线程和多核技术。实际上，在过去的五十年里，工艺的改进对计算机性能提高的贡献远比改进体系结构对性能提高的贡献要大。但时钟频率总是有个界限在那里，所以从事体系结构研究的人要努力了。

1.3.1 计算机性能和性能评价

单从时间的角度观察性能，同样一个程序，计算机执行它时所用的时间越短，它的性能就越好。因此我们可以认为性能是执行时间的倒数。那么如何计算一个程序的执行时间呢？我们有下面的公式：

$$T = I \times CPI \times TPC$$

其中，I (Instructions) 是被执行的指令的总数，CPI (Cycles Per Instruction) 是每条指令执行时所需要的平均的时钟周期数，TPC (Times Per Cycle) 是时钟周期的时间长度。

实际上，全世界的许多研究者都正在为如何减小上述公式中的每一个参数而努力地奋斗着。编译器和体系结构的设计者偏重于减小 I；体系结构和 CPU 的硬件设计者试图减小 CPI；而 CPU 和集成电路的设计者想要减小 TPC。

这三个参数并不是完全独立的，比如 CISC 试图使用复杂的指令减小 I，但会引起 CPI 和 TPC 的增大；而 RISC 可以减小 CPI 甚至 TPC，但会引起 I 的增大。

CPI 的倒数是 IPC (Instructions Per Cycle)，即每个周期能执行多少条指令。当然，IPC 的值是越大越好。自从超标量技术问世之后，人们更多地使用 IPC，而不是 CPI，虽然二者并没有本质的差别。

在讨论计算机性能的改善时，人们经常使用 Amdahl's Law 来计算某个部件的优化对整体性能有多大的改进。假设我们把某个部件优化了 n 倍，即所需时间是原来的 1/n，但执行一个程序时，用到该部件的时间百分比是 r，则整体性能的加速比是

$$S = \frac{P_n}{P_o} = \frac{T_o}{T_n} = \frac{T_o}{T_o \times r/n + T_o \times (1 - r)} = \frac{1}{r/n + (1 - r)}$$

其中，P_n 和 P_o 分别是优化后的整体性能和优化前的整体性能，T_n 和 T_o 分别是优化后程序的执行时间和优化前程序的执行时间。从式中可以看出，即使 $n \to \infty$，S 也有上限 $1/(1 - r)$。假设 $r = 50\%$，那么 S 最大也就是 2。图 1.10 示出 Amdahl's Law 在不同的 n 和 r 下的一些曲线的例子，从中我们可以看出性能加速比的变化趋势。

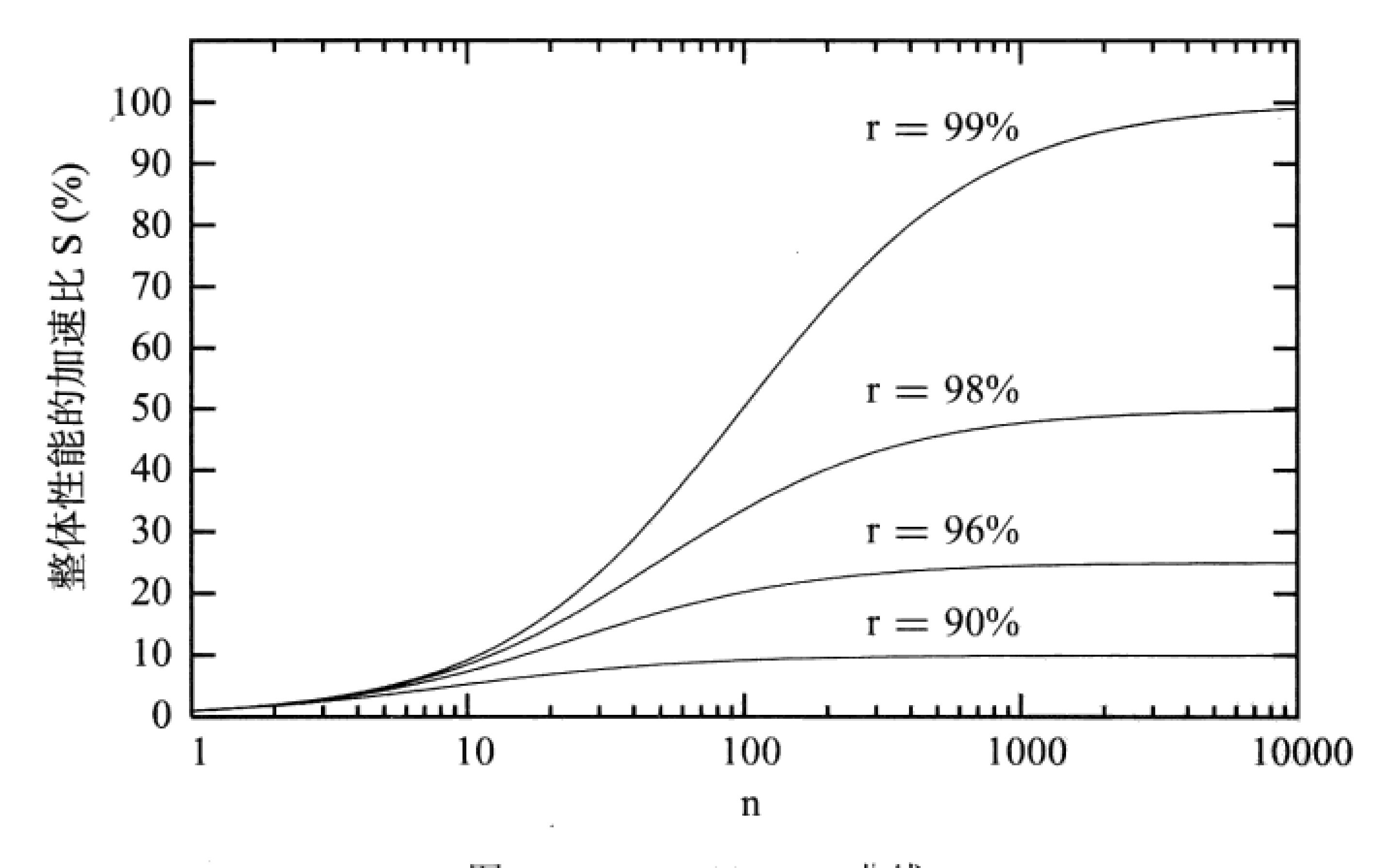
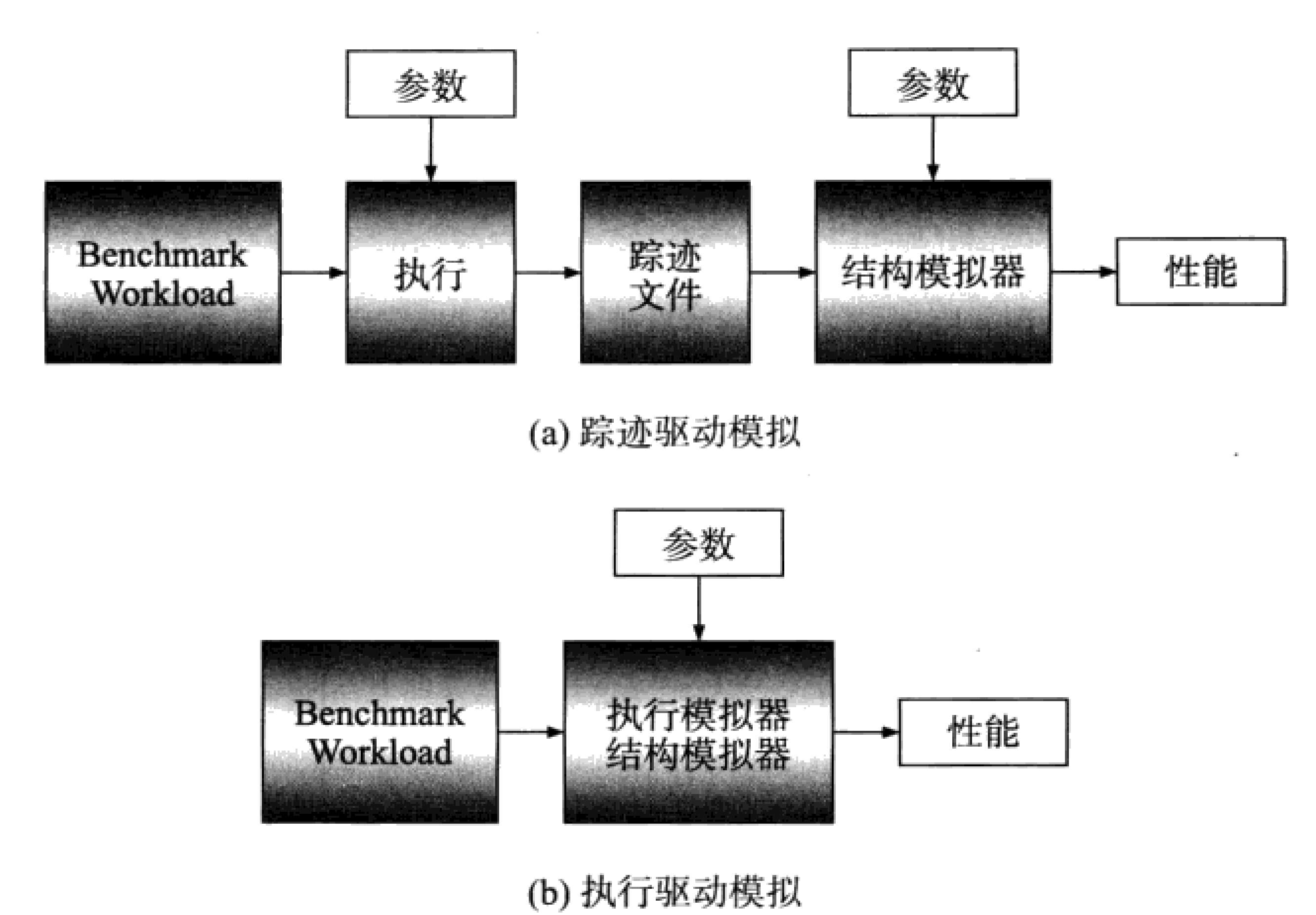


图 1.10 Amdahl's Law 曲线

1.3.2 踪迹驱动模拟和执行驱动模拟

踪迹驱动模拟 (Trace-Driven Simulation) 是在体系结构层次上对新的提案的性能

进行评价的一种方法。它的做法是把实际程序，比如基准程序 (Benchmark)，执行时的踪迹 (比如程序计数器、指令、数据存储器的地址等信息) 记录下来，作为一个文件保存到硬盘上。这个文件将被用来作为一个模拟器的输入。模拟器是用软件的方法实现新的提案并把踪迹作为输入来计算出提案要关心的性能，见图 1.11(a)。



(a) 踪迹驱动模拟

(b) 执行驱动模拟

图 1.11 踪迹驱动模拟和执行驱动模拟

使用这种方法，我们可以对以下内容进行性能评价，比如指令 Cache 和数据 Cache 的结构、ILP、转移预测机制、指令预取缓冲区、各种功能部件的利用率等。通过对模拟结果的分析，我们可以找出系统性能的瓶颈所在并提出改进的方法 (当然还要再进行踪迹驱动模拟)。

踪迹驱动模拟的优点是不涉及具体硬件的实现细节，既可以对还没有实现的新的结构进行评价，也可以对现有的结构进行评价，并且可以改变结构的参数来观察性能的变化趋势 (测试一个已经实现的系统无法做到这一点，因为参数已经固定)。

踪迹驱动模拟所需的踪迹文件会很大，如果你要生成很多的 Benchmark 程序的踪迹，硬盘空间会被迅速占满。当然你可以把这些文件压缩存放，或者只记录部分踪迹。另一种类似的方法不需要存放这些踪迹文件，它就是所谓的执行驱动模拟 (Execution-Driven Simulation)，见图 1.11(b)。执行驱动模拟器的输入不是踪迹文件，而是直接使用实际的基准程序的二进制代码。因此在模拟器中要边模拟执行这些代码 (相当于产生踪迹) 边对提案的结构进行性能评价。

如果一个 Benchmark 程序的执行踪迹要被多次使用，还是一次性地产生一个踪迹文件，使用踪迹驱动模拟方法为上，因为这种方法可以缩短模拟时间 (不必每次都模拟执行基准程序)。

1.3.3 高性能计算机和互联网络

高性能计算机是指那些包含有多个 CPU 或多台计算机的系统。主要分成两类：多处理机 (Multiprocessors) 系统和多计算机 (Multicomputers) 系统。前者共享存储器，又称并行系统 (Parallel Systems)；后者不共享，又称分布式系统 (Distributed Systems)。超级计算机 (Supercomputers) 和伺服器 (Servers) 一般都是共享存储器的并行系统，而用于网格计算 (Grid Computing) 及云计算 (Cloud Computing) 的环境均由分布式系统构成 (其中会有超级计算机)。

互联网络 (Interconnection Networks) 在并行系统中占有重要的位置。它连接所有的 CPU / 存储器板，CPU 与远程存储器 (非本地存储器) 之间的数据传输要经过它才能进行，见图 1.12。

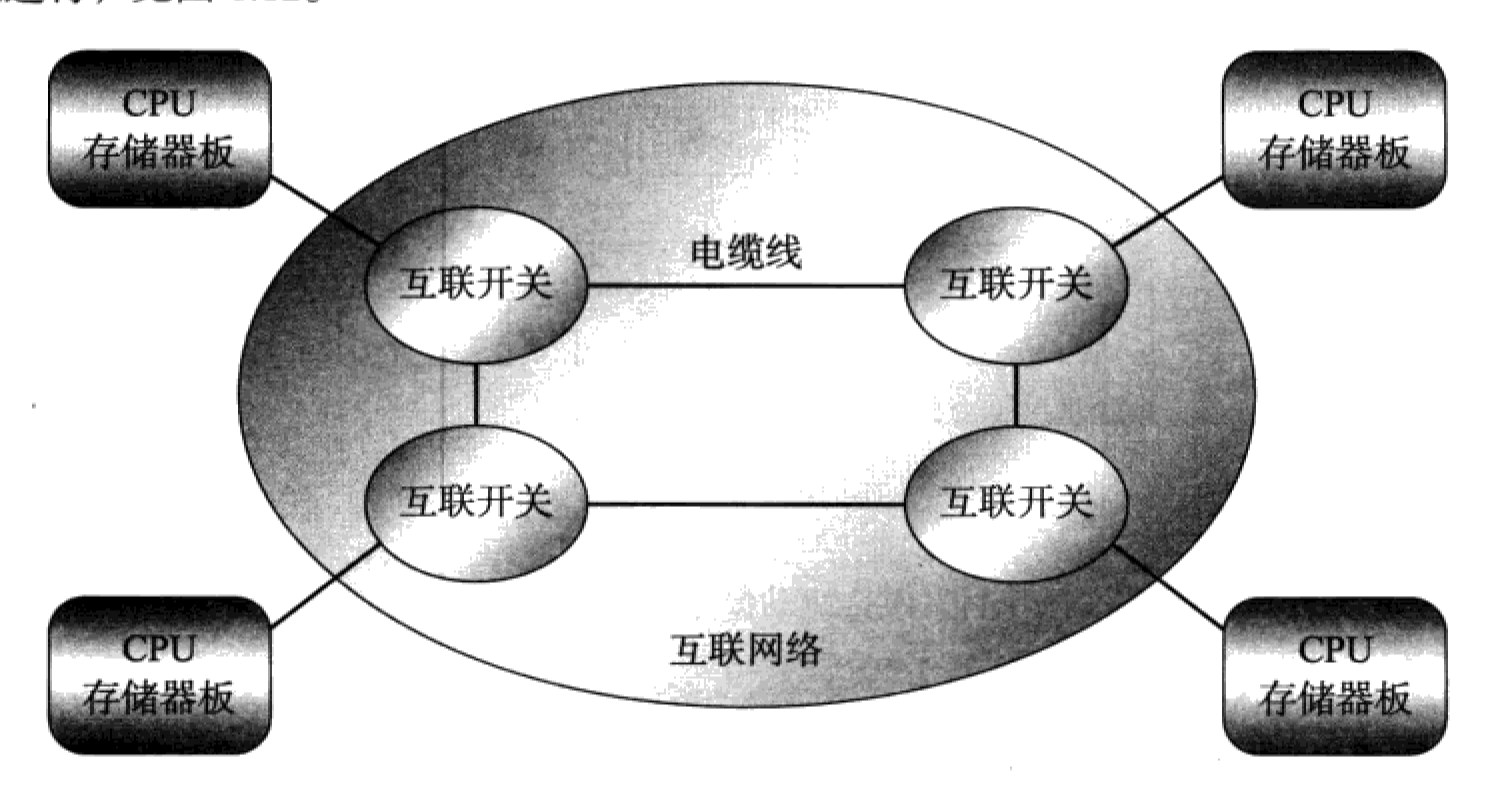


图 1.12 高性能计算机和互联网络

全球 500 强超级计算机的排名每年在 top500.org 网页上公布两次。500 强超级计算机中大致上有 90% 的 CPU 是 Intel 或 AMD 的 x86 系列，其余的 10% 使用 RISC 系列的 CPU。不过在伺服器中，低档的使用 x86 CPU，高档的使用 RISC CPU。

前面提到的 Amdahl's Law 也适用于计算高性能计算机的性能加速比。如果一个程序中有 1% 的代码必须是串行执行的，那么不管 CPU 的数量再多，总的加速比也不会超过 100 (参照图 1.10 中 $r = 99\%$ 的曲线)。因此，为高性能计算机，包括单片的多核 CPU 的计算机，提供并行度高的程序，才能有效地发挥系统的高性能。

1.4 硬件描述语言

硬件描述语言 (Hardware Description Languages) 是设计硬件时使用的语言，比用逻辑图设计方便多了，尤其是当电路规模越大时，这种感觉就越强烈。硬件描述语

言有许多种，其中最常用的有 Verilog HDL 和 VHDL。注意二者虽然都以 V 开头，但不是相同的语言，都有各自的 IEEE 标准。还要注意 Verilog 和 Verilog HDL 是两种不同的东西，前者是逻辑检验器 (Verilog 来自于 Verifying Logic)，后者是语言。

以下是用 Verilog HDL 实现的一个 4 位计数器电路的代码 time_counter_verilog.v。文件名任意，文件扩展名必须是 .v。模块 (module) 名必须与文件名一致。输入信号有两个：enable 和 clk，输出信号一个：my_counter，4 位，是 reg (寄存器) 类型。当 enable 为 1 时，用 clk 的上升沿计数。always 是关键字，posedge 也是关键字，表示上升沿。另外，赋值符号 <= 也可以用 =。你看，即使你不懂很低层的硬件设计，也可以设计硬件，就像不懂汇编和机器码语言也能用 C 或 Java 来开发软件一样。

```
module time_counter_verilog (enable, clk, my_counter);
    input enable, clk;
    output [3:0] my_counter;
    reg [3:0] my_counter;
    always @ (posedge clk) begin
        if (enable)
            my_counter <= my_counter + 4'h1;
    end
endmodule
```

计数器的仿真波形在图 1.13 中给出，是用 Altera 公司的 Quartus II Web Edition 仿真的。my_counter 用十六进制表示。当 enable 为 1 时，计数；当 enable 为 0 时，计数停止。计数值在 clk 的上升沿处立即改变，这是因为我们使用的是功能仿真，没有考虑电路器件的延迟时间。

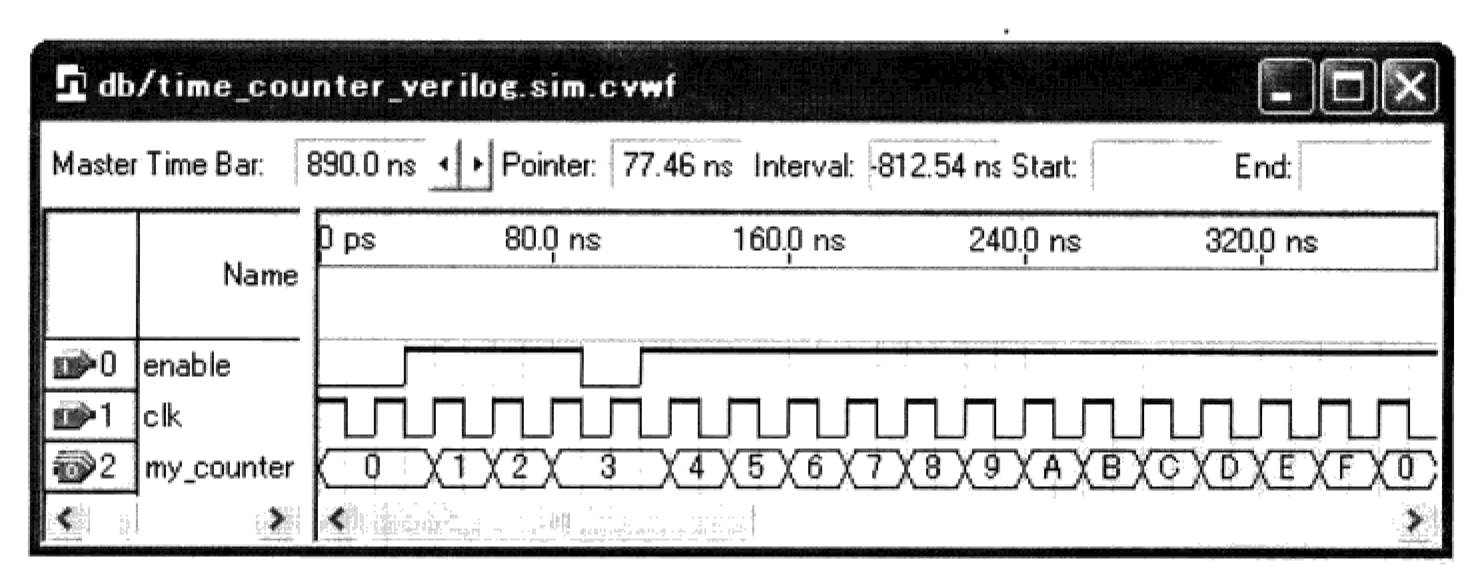


图 1.13 计数器仿真波形

以下是用 VHDL 实现相同的计数器的代码 time_counter_vhdl.vhd，好像比 Verilog HDL 烦琐一些。有人讲 Verilog HDL 像 C 语言，VHDL 像 C++。当然，只从一两个例子还不太能看清楚。

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY time_counter_vhdl IS
    PORT (
        clk        : IN  STD_LOGIC;
        enable     : IN  STD_LOGIC;
        my_counter : OUT INTEGER RANGE 0 TO 15
    );
END time_counter_vhdl;
ARCHITECTURE a_cnt OF time_counter_vhdl IS
BEGIN
    PROCESS (clk)
        VARIABLE cnt : INTEGER RANGE 0 TO 15;
    BEGIN
        IF (clk'EVENT AND clk = '1') THEN
            IF enable = '1' THEN
                cnt := cnt + 1;
            END IF;
        END IF;
        my_counter <= cnt;
    END PROCESS;
END a_cnt;
```

以下是用 Altera 公司的 AHDL 实现相同的计数器的代码 time_counter_ahdl.tdf。很硬，硬到连 D 触发器 DFF 都直接用上了。

```
SUBDESIGN time_counter_ahdl (
    enable, clk       : INPUT;
    my_counter[3..0]  : OUTPUT;
)
VARIABLE
    counter[3..0]     : DFF;
BEGIN
    counter[].clk = clk;
    my_counter[]  = counter[].q;
    IF enable THEN
        counter[] = counter[] + 1;
    ELSE
        counter[] = counter[];
    END IF;
END;
```

Altera Quartus II 也支持 VHDL，当然更支持 AHDL 了，因为那是 Altera 公司自己的语言。VHDL 和 AHDL 的仿真波形与图 1.13 类似，不再给出。上面的 Verilog HDL 代码的例子使用了较高级别的描述风格。实际上，Verilog HDL 的风格可以低到逻辑门 (Gates) 级甚至晶体管级。本书并不详细描述 Verilog HDL 语言本身，而是使用它来设计我们的 CPU 及 I/O 接口和总线控制器。

1.5 习题

1. 列宁说过，忘记过去就意味着背叛。调查一下什么是 DJS-130 和手拨 13 条。
2. 简述 RISC 和 CISC 的主要差别。
3. 详细调查本章给出的 x86 和 MIPS 汇编指令二进制代码中每一位的意义，即它们的指令格式及指令操作码和寄存器的编码，并比较它们的优缺点。
4. 为什么说微程序的方法不易实现指令的流水线操作？
5. 假设我们有两台计算机 M1 和 M2。M1 的主频是 1GHz，M2 是 2GHz。每台计算机的指令都有 4 类，它们的 CPI 分别为 1、2、3 和 4。当同样一个用高级语言编写的程序在两台机器上分别编译并执行时，我们得到下表所列的结果。

计算机	主频	CPI				执行指令数
		1	2	3	4	
M1	1GHz	50%	35%	10%	5%	20 200 000
M2	2GHz	10%	10%	30%	50%	22 000 000

   其中的百分比是执行时每类指令出现的频率。试分别计算该程序在两台机器上的执行时间。如果单从执行时间上考虑，哪一台机器的性能更好？
6. 试计算上题中两台计算机的 MIPS (Million Instructions Per Second)，即每秒能执行多少百万条指令。
7. 如果使用 1 000 000 个 CPU 构建一个并行系统并想得到单 CPU 系统 500 000 倍的性能，那么程序中允许出现的串行执行的代码的比例应该不超过多少才行？
8. 下载并安装 Altera Quartus II Web Edition[3]，然后试着仿真本章最后给出的用三种硬件描述语言设计的计数器代码。

[3]作者在书写本书时使用的是 Quartus II Web Edition 9.0。

第 2 章　逻辑电路及 Verilog HDL 简介

逻辑电路的设计是计算机设计的基础。本章简要介绍逻辑电路设计和基本的 Verilog HDL 的用法。比逻辑电路低一层的是晶体管开关级的电路，本章也简要介绍如何使用 CMOS 晶体管来设计常用的逻辑门。比晶体管开关电路再低一层的是 VLSI 集成电路，其内容已超出了本书的范围。

需要说明的一点是，本书不是 Verilog HDL 的手册或大全。本书的重点是介绍如何使用 Verilog HDL 设计计算机或 CPU。如果读者想要全面了解 Verilog HDL，买一本英文原版或中文翻译的书，或者干脆从网上下载 791 页的 IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-2001 (Revision of IEEE Std 1364-1995), PDF: SS94921 手册[31]。

2.1　基本逻辑门和常用逻辑门

逻辑电路由逻辑门 (Gate) 组成。最基本的逻辑门有 3 个，它们是与门 (AND)、或门 (OR) 和非门 (NOT)。另外还有 4 个常用门：与非门 (NAND)、或非门 (NOR)、异或门 (XOR) 和同或门 (XNOR)。图 2.1 所示的是测试这些逻辑门的电路。这种电路图式的输入方法，英文称做 Schematic Capture，是一种最直观的、但工作量有点大且修改起来比较麻烦的一种输入方法。电路的输入信号有两个：信号 a 和信号 b。输出信号有 7 个：一个门一个。注意，信号名称可以随便取，逻辑门的输入/输出端也不必用线连到相应的输入输出信号端，只要给它们标注上相应的信号名就行。

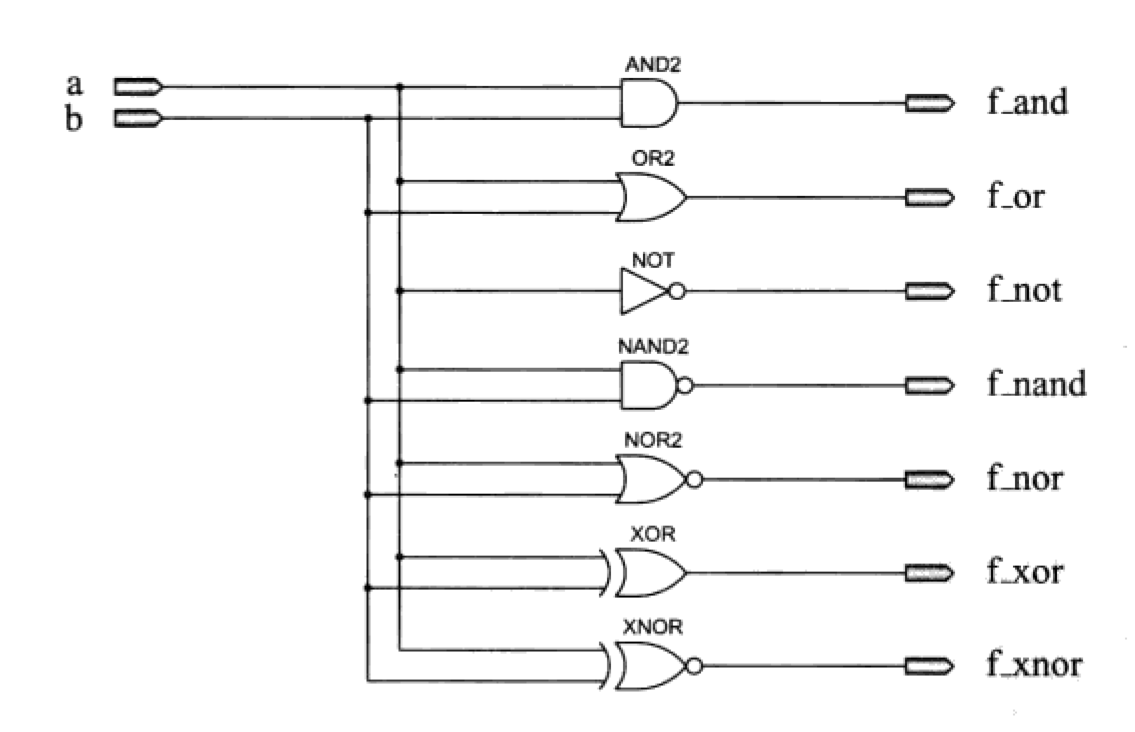


图 2.1　3 个基本逻辑门和 4 个常用逻辑门

另外需要说明的是，本书所使用的逻辑门的符号是国际上比较流行的符号，看起来比较直观。以下是 7 个门的逻辑表达式。注意，这里的“+”代表“或”操作，

而不是相加。其中的 $\overline{a\,b} = \bar{a} + \bar{b}$ 和 $\overline{a + b} = \bar{a}\,\bar{b}$ 被称做迪摩根定理 (DeMorgan's Law)，由出生在印度的英国数学家 Augustus DeMorgan (1806—1871) 最早提出。

$$
\begin{aligned}
&\text{与门 AND:} & f_and &= a \cdot b = a\,b \\
&\text{或门 OR:} & f_or &= a + b \\
&\text{非门 NOT:} & f_not &= \bar{a} \\
&\text{与非门 NAND:} & f_nand &= \overline{a\,b} = \bar{a} + \bar{b} \text{ (DeMorgan's Law)} \\
&\text{或非门 NOR:} & f_nor &= \overline{a + b} = \bar{a}\,\bar{b} \text{ (DeMorgan's Law)} \\
&\text{异或门 XOR:} & f_xor &= a \oplus b = \bar{a}\,b + a\,\bar{b} \\
&\text{同或门 XNOR:} & f_xnor &= a \odot b = \bar{a}\,\bar{b} + a\,b = \overline{a \oplus b}
\end{aligned}
$$

表 2.1 列出了 7 个逻辑门在不同的输入下的输出。输入输出信号的 0 和 1 分别代表信号的电平为低和高。这也是我们对图 2.1 电路进行功能仿真时所期望出现的结果。表 2.1 也有一个好听的名字：真值表 (Truth Table)。

表 2.1 7 个逻辑门的输出 (真值表)

输入		输出						
a	b	f_and	f_or	f_not	f_nand	f_nor	f_xor	f_xnor
0	0	0	0	1	1	1	0	1
1	0	0	1	0	1	0	1	0
0	1	0	1	1	1	0	1	0
1	1	1	1	0	0	0	0	1

现在我们介绍如何使用基本的逻辑门来设计组合电路。逻辑电路有两种类型：组合电路 (Combinational Logic Circuits) 和时序电路 (Sequential Logic Circuits)。组合电路的输出只与当前的输入有关，时序电路不但与当前输入有关，而且与过去的输入 (历史) 有关。

我们通过一个具体的例子来讲解组合电路的设计方法。假设我们的组合电路有一个输出信号 y 和 3 个输入信号 a0、a1 和 s。电路要完成的任务是：当 s 为低电平时，y 与 a0 相同，否则与 a1 相同。我们把这个电路命名为一位二选一多路器，用 mux2x1 表示。设计组合电路的步骤如下：

1) 根据问题的描述画出真值表；
2) 使用卡诺图 (Karnaugh Map) 化简，得到输出信号的逻辑表达式；
3) 根据逻辑表达式画出逻辑电路图；
4) 对电路图进行仿真以检查电路的逻辑是否满足我们的要求。

现在我们画出 mux2x1 的真值表，见表 2.2。图 2.2 所示的是使用卡诺图化简，得到输出信号 y 的逻辑表达式：$y = \bar{s}\ a0 + s\ a1$。图 2.3 是电路图。图 2.4 所示的是仿真结果，其中低电平代表 0，高电平代表 1。

表 2.2 一位二选一多路器真值表

输入信号			输出信号	注 释
s	a1	a0	y	
0	0	0	0	y 与 a0 相同
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	y 与 a1 相同
1	0	1	0	
1	1	0	1	
1	1	1	1	

s \ a1 a0	00	01	11	10
0	0	1	1	0
1	0	0	1	1

$\bar{s}$ a0

s a1



图 2.2 一位二选一多路器的卡诺图化简

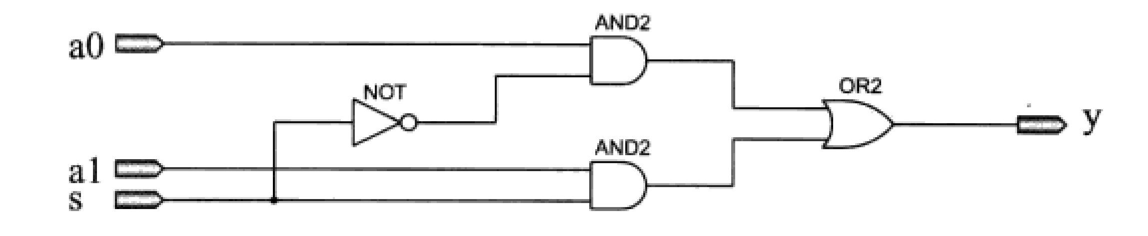


图 2.3 一位二选一多路器的电路图

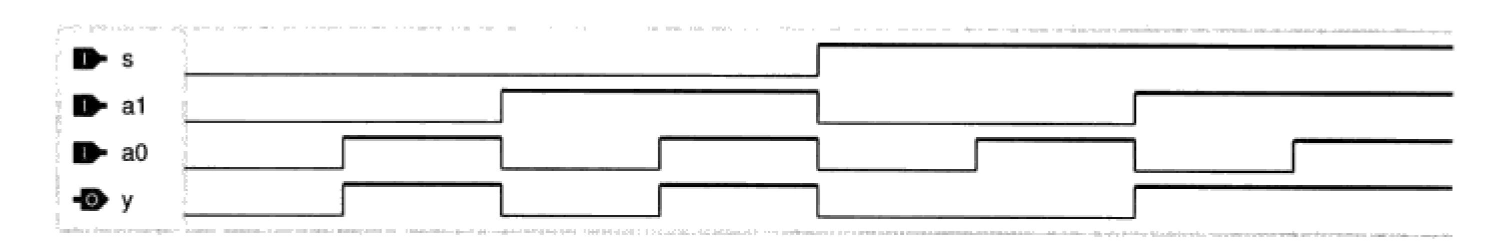


图 2.4 一位二选一多路器的仿真结果

2.2 用 Verilog HDL 实现基本的逻辑操作

Verilog HDL (Hardware Description Language) 是一种设计硬件电路的语言，由 IEEE 完成了对其标准化的工作。Verilog HDL 总体来讲带有 C 语言的风格，是工业界常用的硬件描述语言。另一种具有 C++ 风格的硬件描述语言是 VHDL，也有 IEEE 的标准。比这两种语言层次更高的是 SystemC，一种系统级的描述语言。本书只使用 Verilog HDL。

以下是图 2.1 电路的 Verilog HDL 版本。有了它，我们就不用画图了。Verilog HDL 的一个基本模块由关键字 module 和 endmodule 像括号一样“括”起来。括号的内部描述电路要完成的功能。跟在 module 后面的是模块名，然后是输入输出信号。该段代码的文件名为“gates7_structural.v”，与模块名相同，只是加了扩展名 .v。

```
module gates7_structural (a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,
                          f_xnor);
    input a,b;
    output f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor;
    and  i1 (f_and,a,b);
    or   i2 (f_or,a,b);
    not  i3 (f_not,a);
    nand i4 (f_nand,a,b);
    nor  i5 (f_nor,a,b);
    xor  i6 (f_xor,a,b);
    xnor i7 (f_xnor,a,b);
endmodule
```

Verilog HDL 有多种层次的设计方法。该例使用了比较“低级”的方法，我们这里暂且称其为“逻辑门级”的描述方法。它与图 2.1 的电路完全等同。input 和 output 是关键字：声明哪些是输入信号哪些是输出信号。其余各行的最左单词也是关键字，分别代表上面提到的 7 个逻辑门。它们也是该模块的主体。跟在这些关键字后面的是给相应逻辑门起的名字。括号内的第一个变量是相应逻辑门的输出信号，其余的是输入信号。

模块代码写好后，首先要对它“编译”，看看它是否符合 Verilog HDL 的语法规则。因此，一个 Verilog HDL“编译器”是必不可少的。刚开始接触 Verilog HDL 时，难免会出现许多语法错误。熟练之后，语法错误就慢慢变少了。

即便是语法正确，也不能代表电路能正常工作。为了确认代码是否能完成我们所期望的任务，我们还要对它进行“检查”。检查的方法是给输入信号指定所有可能的输入组合，让模块的主体“计算”出相应的输出。然后我们把计算出的输出与我们所期待的输出进行比较，检查它们是否正确。因此，作为电路的设计者，你必须知道电路的输出应该是什么。如果你不知道，即便电路错了你也查不出来。一般的情况是你原来预想的是正确的，由于代码写错了而计算出错误的结果。这时你要检查代码到底错在哪里。作者也经历过一些有意思的事情，那就是代码是正确的，而原来预想的结果是错误的。也有这种情况出现：仿真结果和预想的一致，但二者都是错误的。这样的错误比较难发现。

计算代码输出的任务由 Verilog HDL“仿真器”完成。因此，我们需要有一个集 Verilog HDL 编译器和仿真器于一身的软件。这样的软件大多由著名的公司开发，例如 Cadence，Synopsys，Mentor Graphics，LogicVision 等。也有一些公司提供免费的软件，例如 Altera 和 Xilinx 等。本书绝大部分代码使用 Altera 公司的 Quartus II 进行编译和仿真。由于基本的 Quartus II 不支持“晶体管开关级”的代码，本书作者从网

上下载了一个非常不错的软件：Icarus Verilog。这是一个源代码公开的软件，编译后生成 iverilog、iverilog-vpi 和 vvp。

我们使用 iverilog 和 vvp 对上述模块进行编译和仿真。为此，我们需要准备一个供测试用的模块，如下所示。文件名为“gates7_structural_test.v”。该测试模块本身对逻辑电路不产生任何影响。测试模块大致由三部分组成：第一部分调用被测试的模块；第二部分用各种输入组合来测试结果；第三部分指定输出文件名。由于第二部分要对输入信号反复赋予不同的值，在模块的开始处要把所有的输入信号声明为“reg”类型。该类型的本意是寄存器 (Register)，但实际上它根本不是真正的寄存器。# 后面的数字是单位时间数，即过了多长时间才开始做后面的工作。测试结果由 $display 语句显示在屏幕上。双引号括起来的是要显示的内容，其中的 %b 代表它后面的信号用二进制格式显示。该例中 $display 语句太多了有点烦，以后我们会介绍不烦的 $monitor 语句。注意，initial 语句只用于仿真。

```
`include "gates7_structural.v"
module gates7_structural_test;
    reg a,b;
    gates7_structural g7 (a,b,f_and,f_or,f_not,
                          f_nand,f_nor,f_xor,f_xnor);
    initial begin
            a = 0; b = 0;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 1; b = 0;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 0; b = 1;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 1; b = 1;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 0; b = 0;
        #1 $finish;
    end
    initial begin
        $dumpfile("gates7_structural.vcd");
```

```
        $dumpvars;
    end
endmodule
```

我们在这段代码的开始处使用了 include 把“gates7_structural.v”调进来，这样在用 iverilog 编译时只要输入文件名“gates7_structural_test.v”就行了。代码中的 $dumpfile 和 $dumpvars 语句用来记录仿真结果，供波形编辑软件 (比如 gtkwave) 使用。仿真结果的文件名为“gates7_structural.vcd”，vcd 是 Value Change Dump 的缩写。它是 ASCII 文本型文件，用来记录各信号值的变化情况。注意，$dumpfile 和 $dumpvars 不影响 $display，如果你不想使用波形编辑软件，这段代码可以删除。

以下是使用 iverilog 和 vvp 在 Linux 环境下对“gates7_structural_test.v”进行编译和仿真时产生的输出。如果不加以指定，iverilog 生成的文件为 a.out。虽然我们使用了 vvp 命令，但 a.out 似乎和 C 程序生成的 a.out 一样，可以直接运行。

```
[yamin@localhost cpu]$ iverilog gates7_structural_test.v
[yamin@localhost cpu]$ vvp a.out
VCD info: dumpfile gates7_structural.vcd opened for output.
 a=0 b=0 f_and=0 f_or=0 f_not=1 f_nand=1 f_nor=1 f_xor=0 f_xnor=1
 a=1 b=0 f_and=0 f_or=1 f_not=0 f_nand=1 f_nor=0 f_xor=1 f_xnor=0
 a=0 b=1 f_and=0 f_or=1 f_not=1 f_nand=1 f_nor=0 f_xor=1 f_xnor=0
 a=1 b=1 f_and=1 f_or=1 f_not=0 f_nand=0 f_nor=0 f_xor=0 f_xnor=1
[yamin@localhost cpu]$
```

顺便提一句，Icarus Verilog 也有 Windows 版本。由于作者喜欢使用 Linux，也就没有去试 Windows 版本的 Icarus Verilog。另外，使用 emacs 编辑 Verilog HDL 源程序时，最好下载并安装一个 Verilog HDL 的 emacs-lisp 程序 verilog-mode.el，这样编辑起 Verilog HDL 程序时感觉非常爽，关键字的颜色和格式设计得相当不错。现在，你可以把仿真输出与表 2.1 相比较，看看结果是不是你想要的。

下面的 Verilog HDL 代码也完成和 gates7_structural.v 同样的任务，但代码的风格与“逻辑门级”代码不同。这里的代码类似于逻辑表达式，我们称其为数据流 (Dataflow) 的描述方法。assign 是关键字，是对信号赋值的意思。记住代码中的操作符号所代表的意义，特别是 AND 和 OR 的操作符号，它们与逻辑表达式中使用的符号是不同的。另外，代码中的括号是必需的，因为取反操作的优先级是最高的。

```
module gates7_dataflow (a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,
                        f_xnor);
    input a,b;
    output f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor;
    assign f_and  =   a & b;
    assign f_or   =   a | b;
    assign f_not  = ~ a;
    assign f_nand = ~(a & b);
    assign f_nor  = ~(a | b);
    assign f_xor  =   a ^ b;
```

```
    assign f_xnor = ~(a ^ b);
endmodule
```

下面是供测试用的代码，我们使用了 $monitor 语句和 $time 变量输出仿真结果。比起 $display 语句，$monitor 语句要方便得多。

```
`include "gates7_dataflow.v"
module gates7_dataflow_test;
    reg a,b;
    gates7_dataflow g7 (a,b,f_and,f_or,f_not,
                        f_nand,f_nor,f_xor,f_xnor);
    initial begin
        $display("time\ta\tb\tand\tor\tnot\tnand\tnor\txor\txnor");
        #0 a = 0; b = 0;
        #1 a = 1; b = 0;
        #1 a = 0; b = 1;
        #1 a = 1; b = 1;
        #1 a = 0; b = 0;
        #1 $finish;
    end
    initial begin
        $monitor ("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",
            $time,a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor);
        $dumpfile("gates7_dataflow.vcd");
        $dumpvars;
    end
endmodule
```

仿真的结果如下所示。注意结果中最左边的数字，它们是 $time 变量的产物。试着改一下测试代码中 # 后面的数字，看看结果会有什么变化。

```
[yamin@localhost cpu]$ iverilog gates7_dataflow_test.v
[yamin@localhost cpu]$ vvp a.out
VCD info: dumpfile gates7_dataflow.vcd opened for output.
time    a       b       and     or      not     nand    nor     xor     xnor
 0:     0       0       0       0       1       1       1       0       1
 1:     1       0       0       1       0       1       0       1       0
 2:     0       1       0       1       1       1       0       1       0
 3:     1       1       1       1       0       0       0       0       1
 4:     0       0       0       0       1       1       1       0       1
[yamin@localhost cpu]$
```

图 2.5 所示的是用 gtkwave 显示仿真结果时的画面。注意 vvp a.out 仿真结果的最后一行 (时间 4) 与第一行重复 (时间 0)，但如果去掉它，用 gtkwave 显示波形，时间 3 的波形出不来。

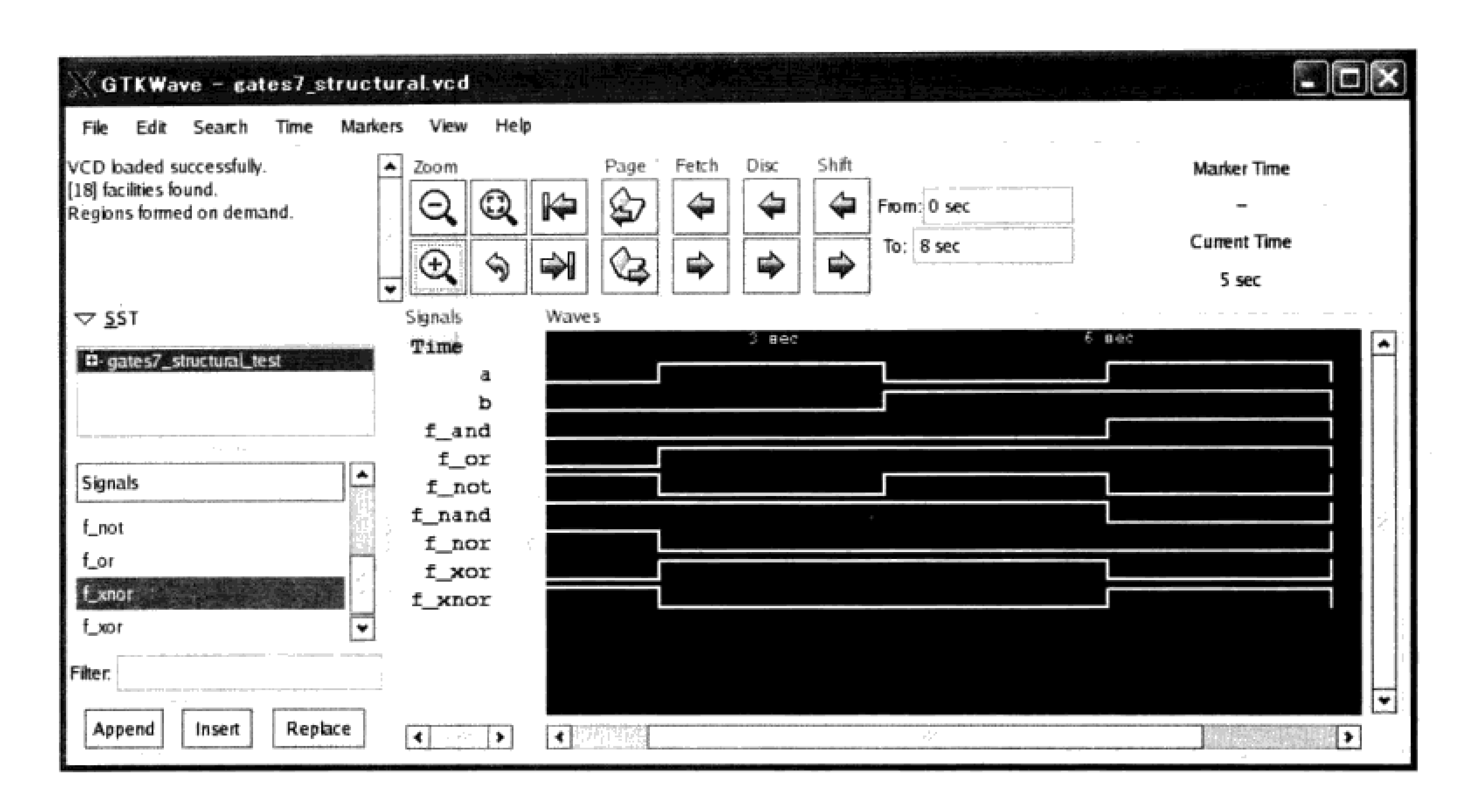


图 2.5 用 gtkwave 显示仿真结果

以上描述的两个 Verilog HDL 版本 (gates7_structural.v 和 gates7_dataflow.v) 虽然完成同样的任务，但完成的方法不同。前者称为结构型描述，后者称为数据流型描述。Verilog HDL 还有其他风格的描述方法，我们将在后面陆续讨论。

2.3 逻辑门的 CMOS 晶体管实现以及晶体管级的 Verilog HDL

以下我们描述如何使用 CMOS 晶体管开关来设计基本的逻辑门。本来这部分内容不包含在逻辑设计中，但了解之后会对电路的优化有帮助。比如我们将会看到与非门 NAND 用的晶体管数量比与门 AND 要少，同样，或非门 NOR 用的晶体管数量比或门 OR 要少，因此在电路中使用 NAND 或者 NOR 会节省晶体管的开销。

2.3.1 CMOS 反向器

CMOS 的全称是 Complementary Metal Oxide Semiconductor，意为互补型金属氧化物半导体，由 PMOS 和 NMOS 两种晶体管“互补”构成。图 2.6 示出了 PMOS 和 NMOS 两种晶体管的符号和由它们组成的 CMOS 反向器电路。

在图 2.6(a) 中，当 PMOS 的栅极输入为低电平时，晶体管导通，漏极的输出与源极相同；在图 2.6(b) 中，当 NMOS 的栅极输入为高电平时，晶体管导通，漏极的输出与源极相同。在图 2.6(c) 中，当输入信号 a 为高电平时，PMOS 晶体管 p1 截止，NMOS 晶体管 n1 导通，输出信号 f 与 gnd (低电平) 相同；当输入信号 a 为低电平时，NMOS 晶体管 n1 截止，PMOS 晶体管 p1 导通，输出信号 f 与 vdd (高电平) 相同。即，$f = \bar{a}$。反向器就是一个 NOT 门。

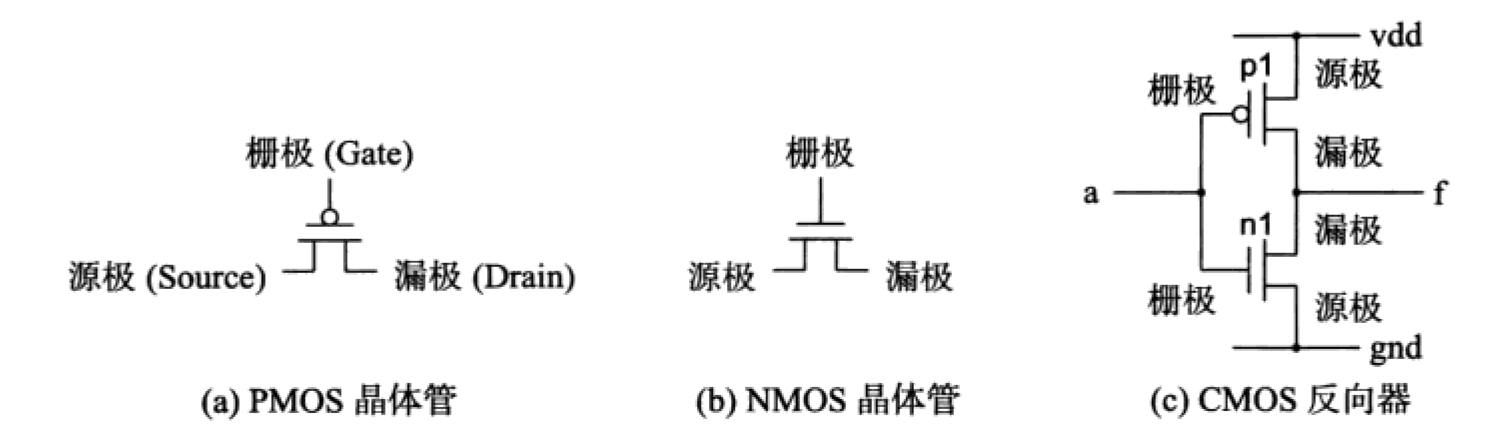


图 2.6 CMOS 反向器电路图

电路电源的低电平用 gnd 表示，高电平用 vdd 表示，有时也用 vcc 表示。一般的约定是使用双极型晶体管 (Bipolar Junction Transistor) 时用 vcc，比如 TTL 电路；使用场效应晶体管 (Field Effect Transistor) 时用 vdd，比如 MOS 电路。随着 TTL 和 MOS 技术的交叉使用，它们之间的界限也变得模糊不清了，比如 CMOS 74HC 系列就使用 vcc 和 gnd 来命名电源和地的管脚。

以下是图 2.6(c) 电路的 Verilog HDL 代码，其中 supply1、supply0、pmos 和 nmos 是关键字。supply1 和 supply0 提供电源和地；pmos 和 nmos 分别代表 PMOS 和 NMOS 晶体管，它们的输入输出信号的次序为漏极、源极和栅极。代码中的双斜杠表示其后面的一行是注释，与 C 语言程序的用法相同。

```
module cmosnot (f,a);
    input a;
    output f;
    supply1 vdd;
    supply0 gnd;
    // pmos name (drain,source,gate);
    // nmos name (drain,source,gate);
    pmos p1 (f,vdd,a);
    nmos n1 (f,gnd,a);
endmodule
```

CMOS 反向器的测试代码如下。

```
`include "cmosnot.v"
module cmosnot_test;
    reg a, b;
    wire f;
    cmosnot not_1 (f,a);
    initial begin
           a = 1;
        #1 a = 0;
        #1 a = 1;
        #1 $finish;
    end
```

```
    initial begin
        $monitor($time,":  a = %b;",a,"  f = %b;",f);
        $dumpfile("cmosnot.vcd");
        $dumpvars;
    end
endmodule
```

下面是 CMOS 反向器的仿真结果。我们可以看出输出与输入电平相反，这就是反向器的作用。

```
[yamin@localhost cpu]$ iverilog cmosnot_test.v
[yamin@localhost cpu]$ vvp a.out
VCD info: dumpfile cmosnot.vcd opened for output.
                   0:  a = 1;  f = 0;
                   1:  a = 0;  f = 1;
                   2:  a = 1;  f = 0;
[yamin@localhost cpu]$
```

2.3.2 CMOS 与非门和或非门

在与非门电路中，只要有一个输入为低电平，输出就为高电平。图 2.7(a) 所示的是二输入 CMOS 与非门电路。当输入信号 a 和 b 同时为高电平时，串联的晶体管 n1 和 n2 均导通，并联的晶体管 p1 和 p2 都截止，这时输出信号 f 为低电平 (与 gnd 相同)。当输入信号 a 和 b 至少有一个为低电平时，串联的晶体管 n1 和 n2 至少有一个截止，而并联的晶体管 p1 和 p2 至少有一个导通，这时输出信号 f 为高电平 (与 vdd 相同)。

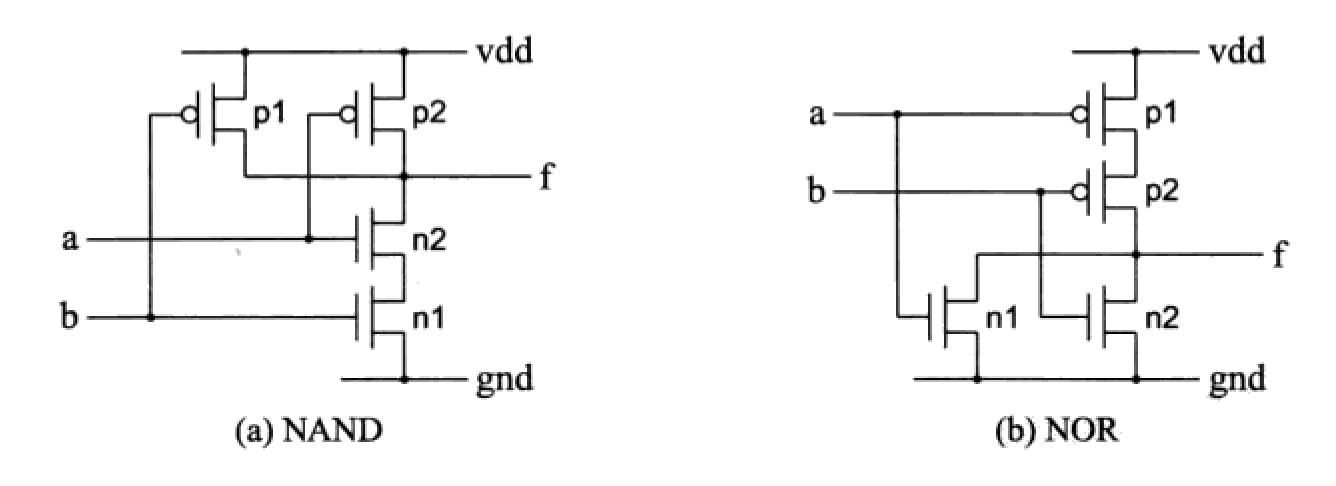


图 2.7 CMOS 与非门 (NAND) 和或非门 (NOR) 电路图

以下是 CMOS 与非门的 Verilog HDL 代码。测试代码我们就不再给出了。

```
module cmosnand (out,ina,inb);
    input ina,inb;
    output out;
    wire   w_n;  // connect the 2 nmos transistors
    supply1 vdd;
    supply0 gnd;
```

```
    pmos p1 (out,vdd,ina);
    pmos p2 (out,vdd,inb);
    nmos n1 (out,w_n,ina);
    nmos n2 (w_n,gnd,inb);
endmodule
```

图 2.7(b) 所示的是二输入 CMOS 或非门电路。当输入信号 a 和 b 同时为低电平时，串联的晶体管 p1 和 p2 均导通，并联的晶体管 n1 和 n2 都截止，这时输出信号 f 为高电平 (与 vdd 相同)。当输入信号 a 和 b 至少有一个为高电平时，串联的晶体管 p1 和 p2 至少有一个截止，而并联的晶体管 n1 和 n2 至少有一个导通，这时输出信号 f 为低电平 (与 gnd 相同)。以下是 CMOS 或非门的 Verilog HDL 代码。

```
module cmosnor (out,ina,inb);
    input ina,inb;
    output out;
    wire   w_p;  // connect the 2 pmos transistors
    supply1 vdd;
    supply0 gnd;
    pmos n1 (out,gnd,ina);
    pmos n2 (out,gnd,inb);
    nmos p1 (w_p,vdd,ina);
    nmos p2 (out,w_p,inb);
endmodule
```

图 2.8 所示的是二输入 CMOS 与门和或门电路。图中的晶体管 p3 和 n3 构成一个非门，即与门由与非门加上非门得到；或门由或非门加上非门得到。因此我们知道与门和或门需要 6 个晶体管，而与非门和或非门只需要 4 个晶体管。如果我们能够在电路中不使用与门和或门，而只使用与非门或者或非门的话，就会减少电路所需晶体管的数量。

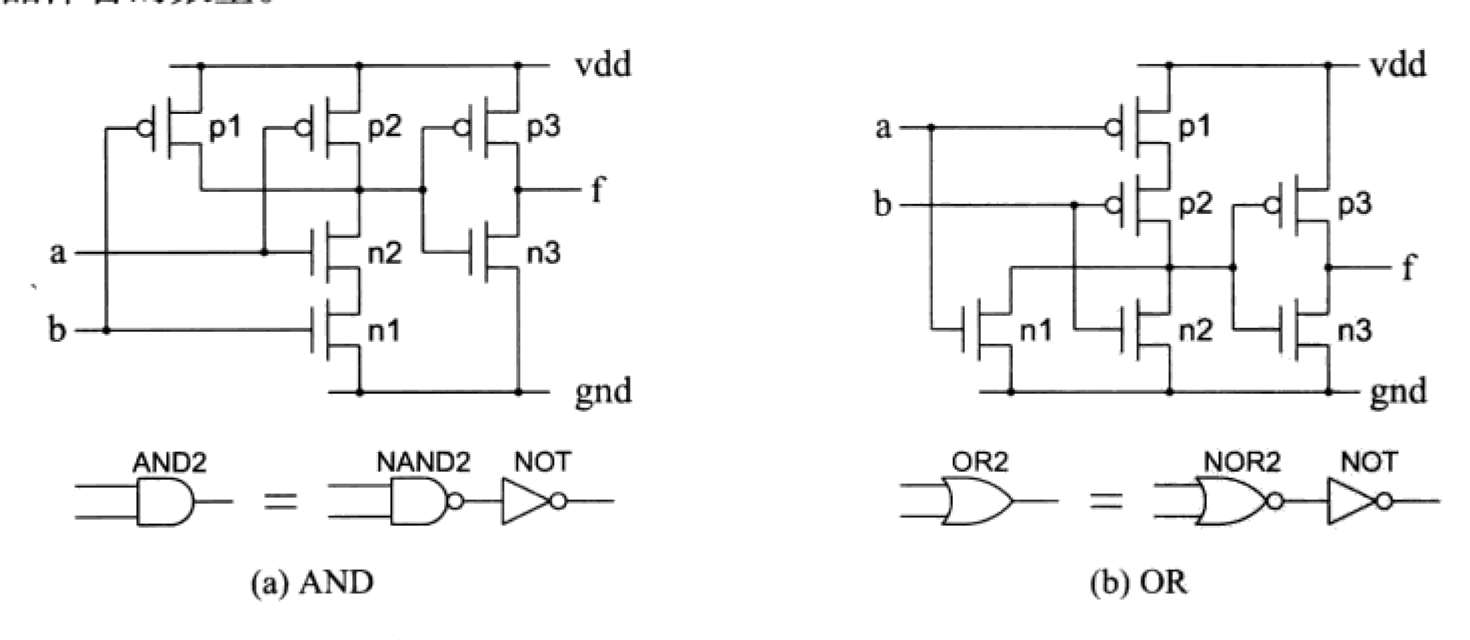


图 2.8 CMOS 与门 (AND) 和或门 (OR) 电路图

图 2.9 所示的是只使用与非门的一位二选一多路器电路。图 2.9(a) 是与图 2.3 相同的一位二选一多路器电路。图 2.9(b) 在与门后面加了两个非门，应该不影响逻辑操

作的结果。图 2.9(c) 是把第二个非门和或门结合在一起的画法。由迪摩根定理，我们得到最后的、如图 2.9(d) 所示的使用与非门的一位二选一多路器的电路图。

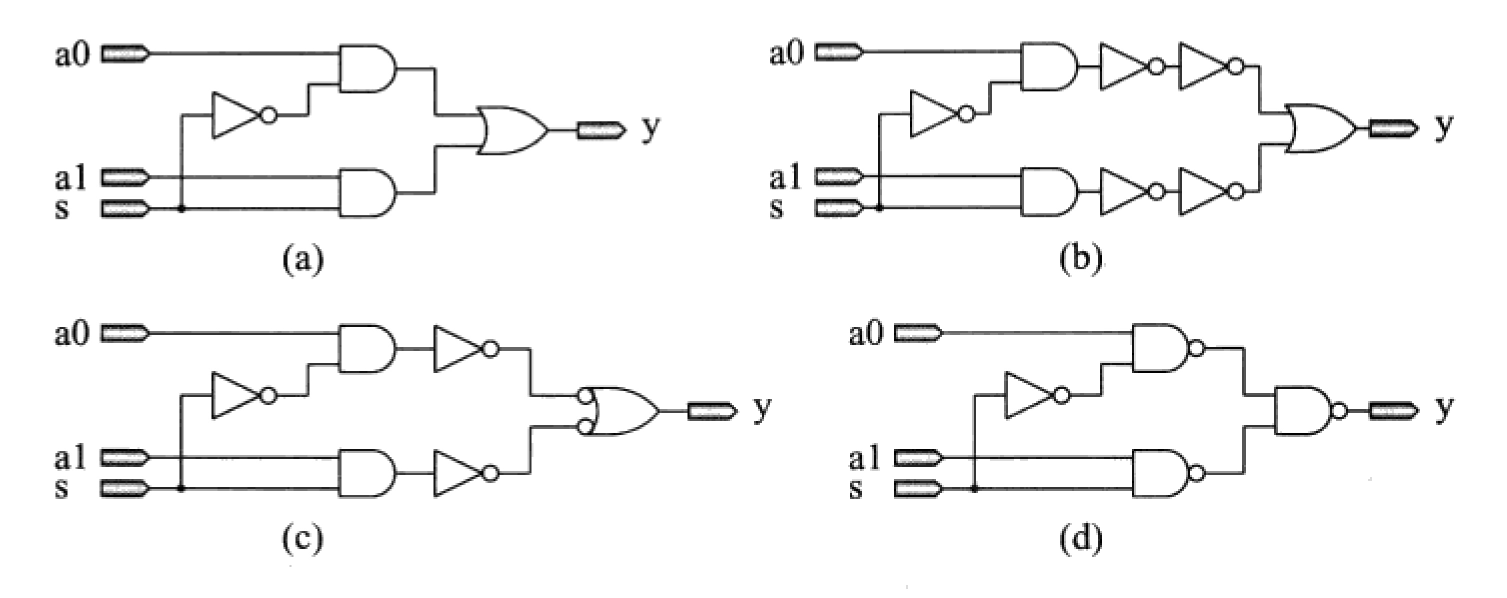


图 2.9 使用与非门 NAND 的一位二选一多路器电路

图 2.9(a) 的二选一多路器需要 20 个晶体管，而图 2.9(d) 只要 14 晶体管就行了。其实图 2.9(d) 中的非门也可以用与非门来替代 (把与非门的两个输入接在一起)。因此只使用与非门，我们可以设计任何形式的电路。在实际的门阵列 (Gate Array) 中，就有只有 NAND 门的电路。

2.4 四种风格的 Verilog HDL 描述

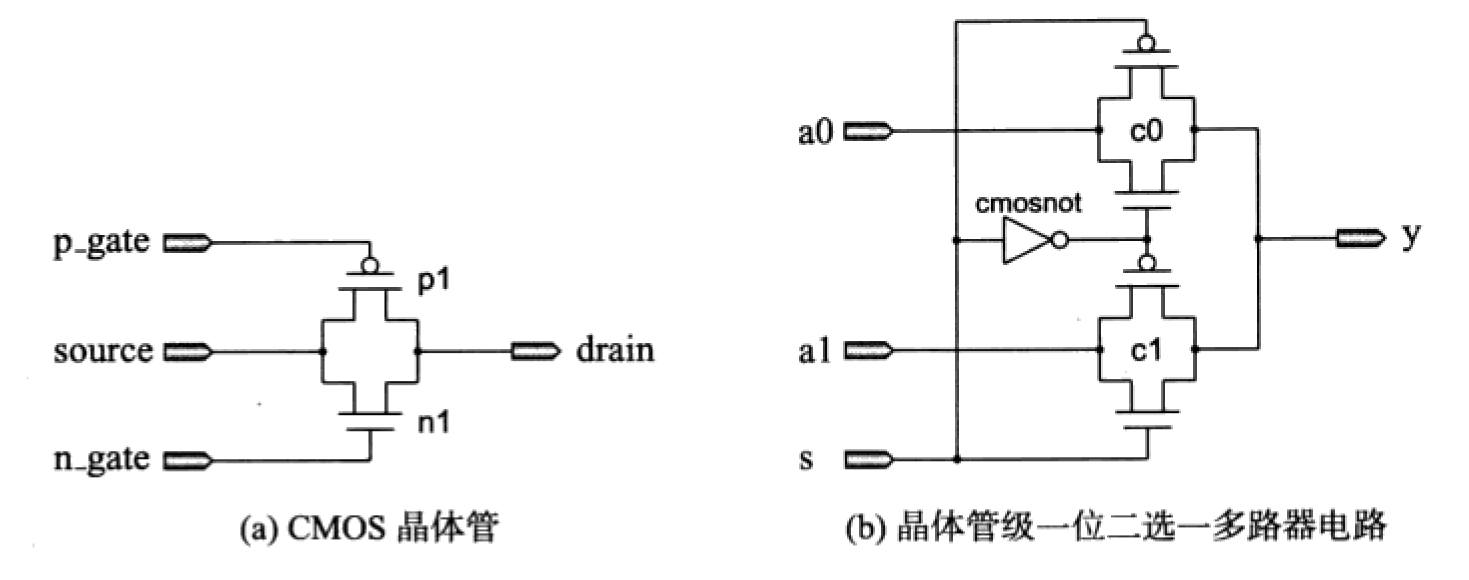
Verilog HDL 提供不同层次的电路设计方法。我们把它分成如下 4 种层次或 4 种风格。层次越低，“硬”件设计的气氛越浓厚。

1) 晶体管开关级 (Switch Level) 的 Verilog HDL 设计；
2) 逻辑门级 (Gate Level) 或结构风格 (Structural Style) 的 Verilog HDL 设计；
3) 寄存器传输级 (RTL) 或数据流风格 (Dataflow Style) 的 Verilog HDL 设计；
4) 功能 (行为) 描述风格 (Behavioral Style) 的 Verilog HDL 设计。

以下我们以一位二选一多路器为例，给出所有这 4 种风格的 Verilog HDL 代码。一位二选一多路器要完成的任务已经在前两节介绍过了。

2.4.1 晶体管开关级的 Verilog HDL

图 2.10 是 CMOS 晶体管开关级的一位二选一多路器的电路。图 2.10(a) 是一个 CMOS 开关。当输入信号 p_gate 为低电平且 n_gate 为高电平时，PMOS 晶体管 p1 和 NMOS 晶体管 n1 均导通，输出信号 drain 等于输入信号 source。图 2.10(b) 使用两个这样的开关实现二选一。图中虽然画了个非门，但它是 CMOS 的反向器 (cmosnot)。当输入信号 s 为低电平时，CMOS 开关 c0 导通，c1 截止，y = a0；否则，c0 截止，c1 导通，y = a1。



(a) CMOS 晶体管　　(b) 晶体管级一位二选一多路器电路

图 2.10　CMOS 一位二选一多路器电路

以下是晶体管开关级的一位二选一多路器的 Verilog HDL 代码，实现图 2.10(b) 的电路，其中 cmosnot not1 (sn, s) 调用已经描述过的 CMOS 的反向器，两个 cmoscmos 调用 CMOS 开关 (图 2.10(a)，源代码留给读者练习)。

```
module mux2x1_cmos (a0,a1,s,y);
    input s,a0,a1;
    output y;
    wire   sn;
    cmosnot not1 (sn,s);
    // cmoscmos (drain,source,n_gate,p_gate);
    cmoscmos c0 (y,a0,sn,s);
    cmoscmos c1 (y,a1,s,sn);
endmodule
```

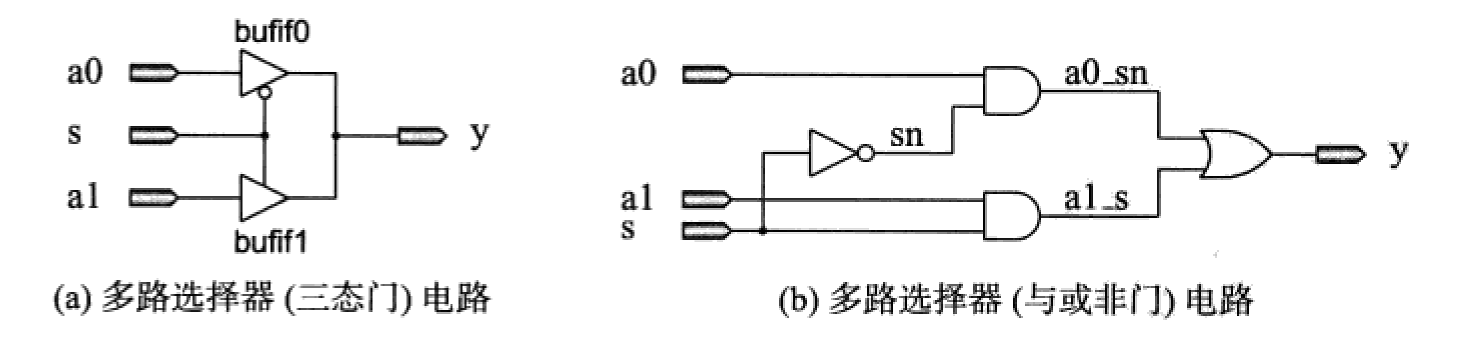
测试代码及仿真结果如下所示。从结果看出，它确实完成了二选一的任务。

```
`include "mux2x1_cmos.v"
`include "cmoscmos.v"
`include "cmosnot.v"
module mux2x1_cmos_test;
    reg s,a0,a1;
    mux2x1_cmos mux2x1 (a0,a1,s,y);
    initial begin
           s = 0; a1 = 0; a0 = 0;
        #1 s = 0; a1 = 0; a0 = 1;
        #1 s = 0; a1 = 1; a0 = 0;
        #1 s = 0; a1 = 1; a0 = 1;
        #1 s = 1; a1 = 0; a0 = 0;
        #1 s = 1; a1 = 0; a0 = 1;
        #1 s = 1; a1 = 1; a0 = 0;
        #1 s = 1; a1 = 1; a0 = 1;
        #1 s = 0; a1 = 0; a0 = 0;
        #1 $finish;
```

```
    end
    initial begin
        $monitor($time,":\ts = %b\ta1 = %b\ta0 = %b\ty = %b",
                 s,a1,a0,y);
        $dumpfile("mux2x1_cmos.vcd");
        $dumpvars;
    end
endmodule
[yamin@localhost cpu]$ iverilog mux2x1_cmos_test.v
[yamin@localhost cpu]$ vvp a.out
VCD info: dumpfile mux2x1_cmos.vcd opened for output.
                    0:   s = 0   a1 = 0   a0 = 0   y = 0
                    1:   s = 0   a1 = 0   a0 = 1   y = 1
                    2:   s = 0   a1 = 1   a0 = 0   y = 0
                    3:   s = 0   a1 = 1   a0 = 1   y = 1
                    4:   s = 1   a1 = 0   a0 = 0   y = 0
                    5:   s = 1   a1 = 0   a0 = 1   y = 0
                    6:   s = 1   a1 = 1   a0 = 0   y = 1
                    7:   s = 1   a1 = 1   a0 = 1   y = 1
                    8:   s = 0   a1 = 0   a0 = 0   y = 0
[yamin@localhost cpu]$
```

2.4.2 逻辑门级的 Verilog HDL

我们给出两种逻辑门级的二选一多路器电路：一种使用三态门 (图 2.11(a))，另一种使用普通的与或非门 (图 2.11(b))。所谓三态门，是指输出有三个状态：一个是 0，一个是 1，还有一个是所谓高阻 (High Impedance) 状态。高阻状态可以简单地理解为只有一条线在那儿浮空，没有任何逻辑门驱动它。图中的输入信号 s 为低电平时，三态门 bufif0 的输出等于输入信号 a0，此时的三态门 bufif1 输出高阻。如果 s 为高电平， bufif0 输出高阻而 bufif1 的输出与输入信号 a1 相同。



(a) 多路选择器 (三态门) 电路　　(b) 多路选择器 (与或非门) 电路

图 2.11　逻辑门级的多路选择器电路

以下的 Verilog HDL 代码实现三态门级的一位二选一多路器，其中 bufif0 和 bufif1 是两个三态门。测试代码与 CMOS 晶体管开关级的测试代码类似，此处就不再列出了。

```
module mux2x1_3s (a0,a1,s,y);
    input s,a0,a1;
    output y;
    bufif0 b0 (y,a0,s);
    bufif1 b1 (y,a1,s);
endmodule
```

普通与或非门的实现代码如下所示，它与图 2.11(b) 的电路完全等价。

```
module mux2x1_gate (a0,a1,s,y);
    input s,a0,a1;
    output y;
    not i0 (sn,s);
    and i1 (a0_sn,a0,sn);
    and i2 (a1_s, a1,s );
    or  i3 (y,a0_sn,a1_s);
endmodule
```

2.4.3 数据流风格的 Verilog HDL

与晶体管开关级和逻辑门级不同，数据流风格的 Verilog HDL 不指定任何特定的器件。它的主要特征是使用 assign 语句对变量赋值，有些类似于逻辑表达式的风格。赋值语句的特点是，等式右边的输入变量一旦发生变化，立即反映到等式左边的输出变量。使用 assign 语句时，对一个输出变量只能赋值一次。

以下是两种数据流风格的 Verilog HDL 代码，实现一位二选一多路器。首先是逻辑表达式风格的代码：

```
module mux2x1_dataflow1 (a0,a1,s,y);
    input s,a0,a1;
    output y;
    assign y = ~s & a0 | s & a1;
endmodule
```

第二个版本使用类似于 C 语言的 if-else 问号形式的语句。问号表示要测试输入信号 s，如果它为 1，则把冒号前面的 a1 赋给 y；否则，把冒号后面的 a0 赋给 y。该例好像具有功能描述风格，属于疑似病例，有待确诊。

```
module mux2x1_dataflow2 (a0,a1,s,y);
    input s,a0,a1;
    output y;
    assign y = s? a1 : a0;
endmodule
```

图 2.12 示出的是使用 Altera 公司的 Quartus II 软件对上述两种代码进行编译及逻辑综合后产生的 RTL 级别的逻辑图。第一个版本的代码产生与逻辑门级类似的逻辑图，而第二个版本产生的只是多路器的符号，它里面的内容是什么，我们无法从外面看到。注意我们把两个版本的代码合在一起了，用 y0 和 y1 区分两个输出。在实际的电路设计过程中，我们并不需要由 Verilog HDL 代码来生成逻辑图。

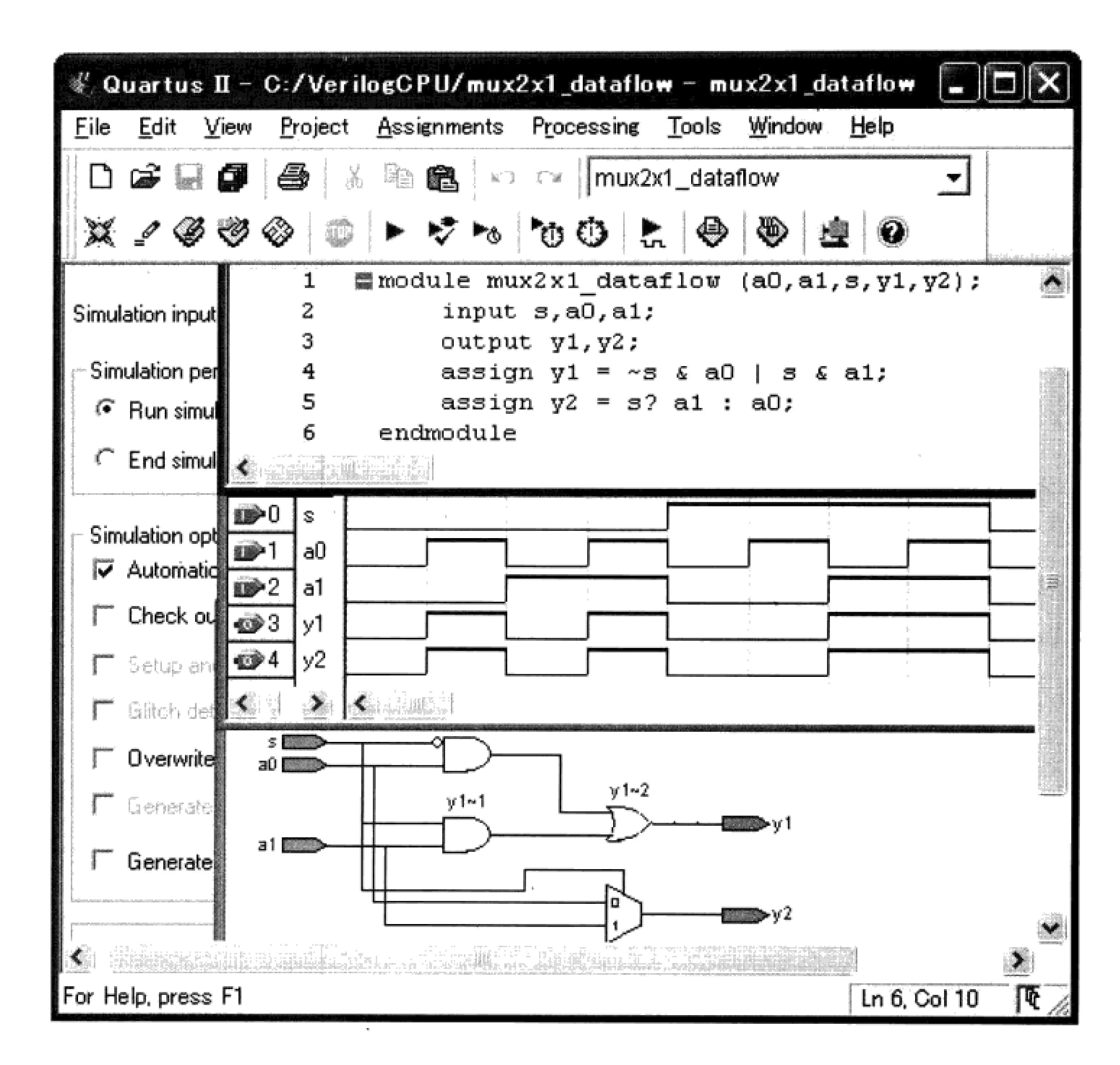


图 2.12　Quartus II 生成的数据流风格的 Verilog HDL 的电路图

2.4.4　功能描述风格的 Verilog HDL

功能 (或行为) 描述风格的 Verilog HDL 是 Verilog HDL 中层次最高的描述方法，它的主要特点是靠“算法”实现电路的操作。对于不太熟悉或不太喜欢低层次硬件逻辑图的人来讲，功能描述风格的 Verilog HDL 是一种最佳选择。以下给出 6 个版本实现一位二选一多路器。虽说是 6 个版本，其实只有两类：always 和 function。

第一个版本使用 always 和 if-else 语句对输出变量赋值。如果没有 always 而直接使用 if-else 语句，就会出现语法错误。always 后面括号中的是输入变量，所有在 always 内部使用的输入变量都要在括号中列出。这是 Verilog HDL 1995 版本的规定，2001 年新的版本只要写一个星号就行了。另外，在 always 内部被赋值的变量都要声明为 reg 类型，如果声明为 wire 类型，编译就会报错。尽管声明为 reg 类型，如果在 always 内部的算法对所有可能的情况都加以描述，最后产生的电路中不会包含任何寄存器。一旦有一种情况没考虑到，将会生成时序电路，即电路里面会有寄存

器。多路器是纯组合电路，不应包含任何寄存器。注意 always 内部不能使用 assign。

```
module mux2x1_behavioral_if_else (a0,a1,s,y);
    input s,a0,a1;
    output y;
    reg    y; // 'y' cannot be a net
    always @ (s or a0 or a1) begin
        if (s) begin
            y = a1;
        end else begin
            y = a0;
        end
    end
endmodule
```

第二个版本与上面的类似，只是不管三七二十一，先把输出变量都赋了值再说。这种做法一般能保证产生的是组合电路。

```
module mux2x1_behavioral_if (a0,a1,s,y);
    input s,a0,a1;
    output y;
    reg    y; // 'y' cannot be a net
    always @ (s or a0 or a1) begin
        y = a0;
        if (s) begin
            y = a1;
        end
    end
endmodule
```

第三个版本还是使用 always 语句，但在 always 内部，我们使用了 case 语句。同样，所有的 case 如果都涉及了，生成的电路为组合电路，否则为时序电路。一撇左边的 1 表示是一位；右边的 b 表示用二进制格式给出变量的值。其他的还有：d 表示十进制、h 表示十六进制。

```
module mux2x1_behavioral_case_all (a0,a1,s,y);
    input s,a0,a1;
    output y;
    reg    y; // 'y' cannot be a net
    always @ (s or a0 or a1) begin
        case (s)
            1'b0: y = a0;
            1'b1: y = a1;
        endcase
    end
endmodule
```

与上面的版本类似，第四个版本也使用 case 语句，但用了 default 语句。所有没有涉及的 case，都归类到 default 语句中，即它是默认的 case。这种做法也能保证产生的是组合电路。

```
module mux2x1_behavioral_case_default (a0,a1,s,y);
    input s,a0,a1;
    output y;
    reg    y; // 'y' cannot be a net
    always @ (s or a0 or a1) begin
        case (s)
            1'b0: y = a0;
            default: y = a1; // default
        endcase
    end
endmodule
```

第五个版本的代码有些不同。输出信号 y 用数据流风格的语句赋值，但所赋的值的内容由函数 sel 产生。注意函数 function 代码的写法。它不用括号，也不声明输出变量。实际上函数名 sel 就相当于输出变量。函数 function 的内部变量名任意，但要与调用它的语句中的变量相对应。本例中在 function 的内部使用的是 case 语句。

```
module mux2x1_behavioral_function_case_all (a0,a1,s,y);
    input s,a0,a1;
    output y;
    assign y = sel (a0,a1,s);
    function sel;
        input a,b,c; // note the order
        case (c)
            1'b0: sel = a;
            1'b1: sel = b;
        endcase
    endfunction
endmodule
```

最后一个版本也是使用 function，但在 function 内部用了 if-else 语句。

```
module mux2x1_behavioral_function_if_else (a0,a1,s,y);
    input s,a0,a1;
    output y;
    assign y = sel (a0,a1,s);
    function sel;
        input a,b,c; // note the order
        if (c) sel = b;
        else   sel = a;
    endfunction
endmodule
```

总之，功能描述风格的代码可以用 always 或 function 实现。在 always 和 function 内部，可以使用 case 语句和 if-else 语句。所有的 6 个版本均会输出与图 2.12 中的 y2 信号相同的电路图。以上的 Verilog HDL 代码实现的都是组合电路，我们将在 2.6 节介绍用 Verilog HDL 实现时序电路。在以后各章的叙述中，我们将给出更多的实例来说明如何用它来实现算法和设计 CPU。

2.5 常用的组合电路及其设计

本节介绍常用的组合电路并给出它们的 Verilog HDL 代码。这些内容都是我们以后设计 CPU 时要用到的。

2.5.1 多路选择器设计

本小节介绍多路选择器的设计。虽然我们已经介绍过多路器的设计了，但那只是二选一，并且是一位的。以下是 32 位的二选一多路器的代码，其中 [31:0] 表示 a0 和 a1 都是 32 位的输入。每一位都有自己的名字，例如：a0[31]、a0[15]、a0[0] 等。注意选择信号 s 只有一位，当它为 1，选中 a1 的所有 32 位，由同样是 32 位的输出信号 y 送出；当 s 为 0 时，选中 a0。

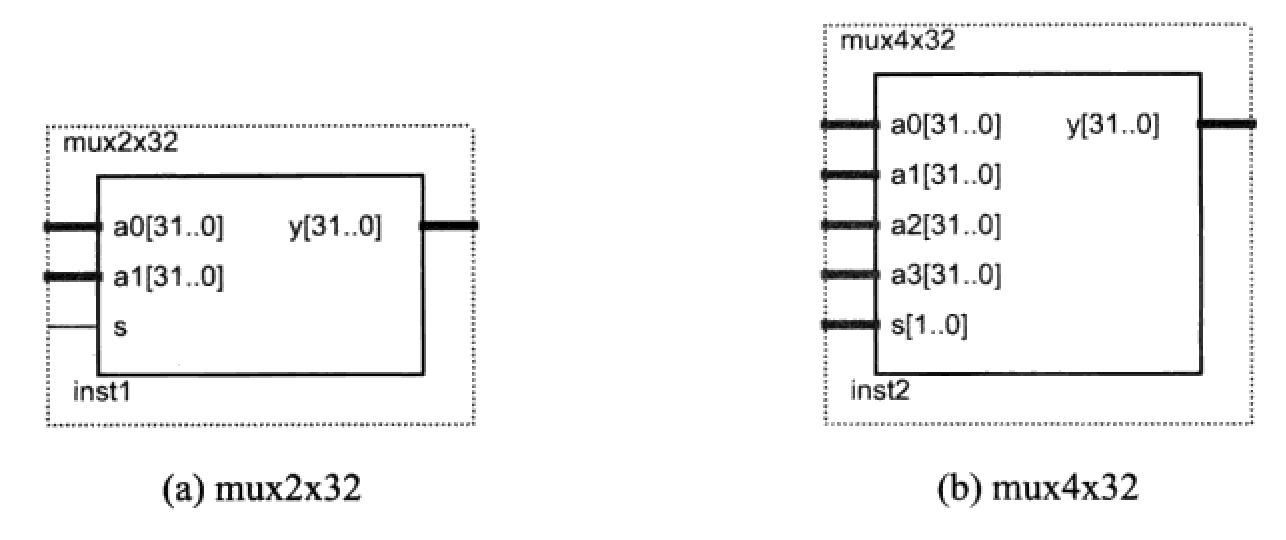
```
module mux2x32 (a0,a1,s,y);
    input [31:0] a0,a1;
    input        s;
    output [31:0] y;
    assign        y = s? a1 : a0;
endmodule
```

以下代码实现 32 位四选一多路器。这时需要两位选择信号：当 s = 00 时，y = a0；s = 01 时，y = a1；s = 10 时，y = a2；s = 11 时，y = a3。我们在本例中使用的是 function，请读者用 always 语句或者嵌套的“? :”assign 语句实现它。

```
module mux4x32 (a0,a1,a2,a3,s,y);
    input [31:0] a0,a1,a2,a3;
    input [1:0]  s;
    output [31:0] y;
    function [31:0] select;
        input [31:0] a0,a1,a2,a3;
        input [1:0]  s;
        case (s)
            2'b00: select = a0;
            2'b01: select = a1;
            2'b10: select = a2;
            2'b11: select = a3;
        endcase
    endfunction
```

```
    assign y = select(a0,a1,a2,a3,s);
endmodule
```

图 2.13 是我们用 Quartus II 对以上代码生成的逻辑符号图。如果信号只有一位，用细线表示，否则用粗线。另外， Quartus II 对信号的命名方法与 Verilog HDL 略有不同，找找看，哪儿不同。如果不使用电路图输入，则没必要产生符号图。



(a) mux2x32　　(b) mux4x32

图 2.13　Quartus II 产生的 32 位二选一多路器和四选一多路器的符号图

2.5.2　译码器设计

一个 m-2^m 译码器 (Decoder) 的功能如下所述。设有一位输入信号 ena、m 位输入信号 n[m−1:0] 和 2^m 位输出信号 e[2^m−1:0]。如果 ena = 1，则输出信号 e[n] = 1，其余为 0；如果 ena = 0，所有的输出信号为 0。表 2.3 所示的是一个 3-8 译码器的真值表。

表 2.3　3-8 译码器的真值表

输入				输出							
ena	n[2]	n[1]	n[0]	e[7]	e[6]	e[5]	e[4]	e[3]	e[2]	e[1]	e[0]
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0
0	x	x	x	0	0	0	0	0	0	0	0

由真值表，我们有如图 2.14 所示的逻辑电路图。这里省略了写逻辑表达式的步骤。注意，电路图没有连线，而是采用标注信号名的方法，这样画起图来变得简单且容易检查。另外还要注意，我们对输入输出信号的命名用了与 Verilog HDL 相同的

规则，即数字之间用冒号，而 Quartus II 用两个点，见图 2.13。逻辑图不是重点，重点是 Verilog HDL 代码。

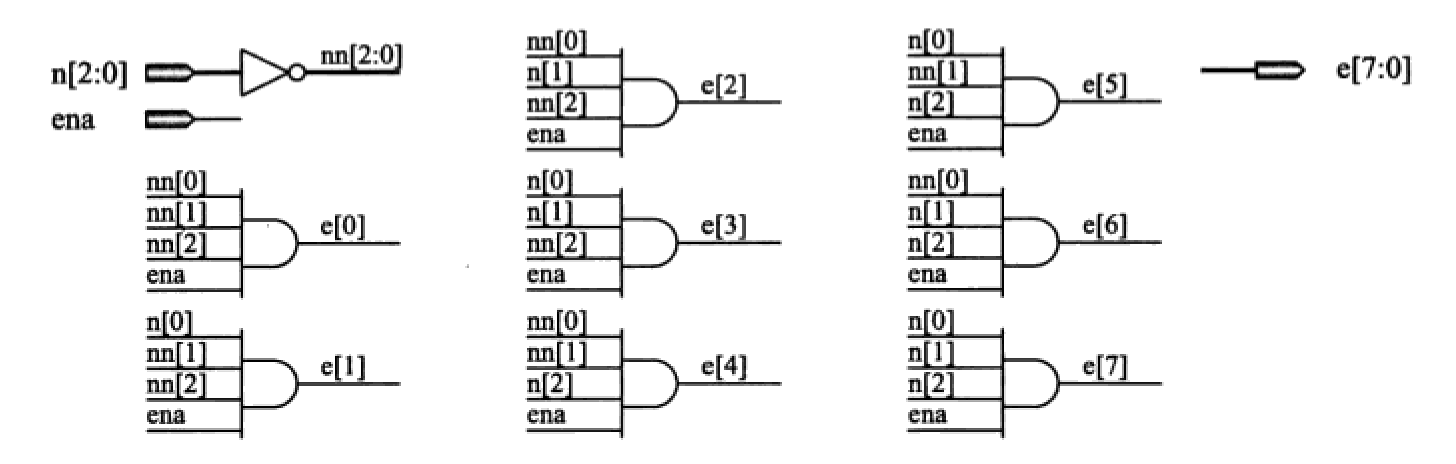


图 2.14 3-8 译码器逻辑电路图

以下的 Verilog HDL 代码实现 3-8 译码器电路。如果我们使用 case 语句，则要写很多行代码。我们这里用了类似于数组元素赋值的方法：先把所有的输出赋 0，然后再只对其中的一位赋 1 或 0。还有更简单的，见本章习题。

```
module decoder3e (n,ena,e);
    input [2:0] n;
    input       ena;
    output [7:0] e;
    reg [7:0]    e;
    always @ (ena or n) begin
        e = 8'b0;
        e[n] = ena;
    end
endmodule
```

下面的测试代码中也用了我们还没有描述过的语句：for 循环。循环变量 i 要声明为 integer，即整数。你看，与 C 语言多么相似。

```
`include "decoder3e.v"
module decoder3e_test;
    reg [2:0] n;
    reg       ena;
    wire [7:0] e;
    integer    i;
    decoder3e dec3e (n,ena,e);
    initial begin
        #0 $display("time\tena\tn\te");
        #0 ena = 1; n = 0;
        for (i = 1; i < 8; i = i + 1) begin
            #1 n = i;
        end
        #1 ena = 0;
```

```
        #1 ena = 1; n = 0;
        #1 $finish;
    end
    initial begin
        $monitor("%1d\t%b\t%b\t%b",$time,ena,n,e);
        $dumpfile("decoder3e.vcd");
        $dumpvars;
    end
endmodule
```

仿真结果如下，与表 2.3 所期待的输出相同，一看就知道是我想要的。

```
[yamin@localhost cpu]$ iverilog decoder3e_test.v
[yamin@localhost cpu]$ vvp a.out
VCD info: dumpfile decoder3e.vcd opened for output.
time    ena     n       e
0       1       000     00000001
1       1       001     00000010
2       1       010     00000100
3       1       011     00001000
4       1       100     00010000
5       1       101     00100000
6       1       110     01000000
7       1       111     10000000
8       0       111     00000000
9       1       000     00000001
[yamin@localhost cpu]$
```

2.5.3 32 位移位器设计

32 位移位器对 32 位二进制数左移或右移，移位位数可在 0 ~ 31 之间自由选择，右移时可选择逻辑右移 (高位填 0) 或算术右移 (高位符号扩展)。以下是 3 种移位操作的例子，它们分别对原始数据进行左移、逻辑右移和算术右移。

原始数据：	1111_1111_0000_0000_0000_0000_1111_1111
左移 8 位：	0000_0000_0000_0000_1111_1111_**0000_0000**
逻辑右移 8 位：	**0000_0000**_1111_1111_0000_0000_0000_0000
算术右移 8 位：	**1111_1111**_1111_1111_0000_0000_0000_0000

图 2.15 所示的是 32 位左移移位器电路图。图中有 5 个 32 位二选一多路器。控制移位位数的输入信号 sa[4:0] 有 5 位。5 位全为 0 时表示移 0 位，全为 1 时表示移 31 位。即，sa[0] = 1 时移 1 位，sa[1] = 1 时移 2 位，sa[2] = 1 时移 4 位，sa[3] = 1 时移 8 位，sa[4] = 1 时移 16 位。图中的小小长方形是二选一多路器，sa 被用作多路器的选择信号，0 选左 1 选右。注意看 sa 中每一位的厉害程度。

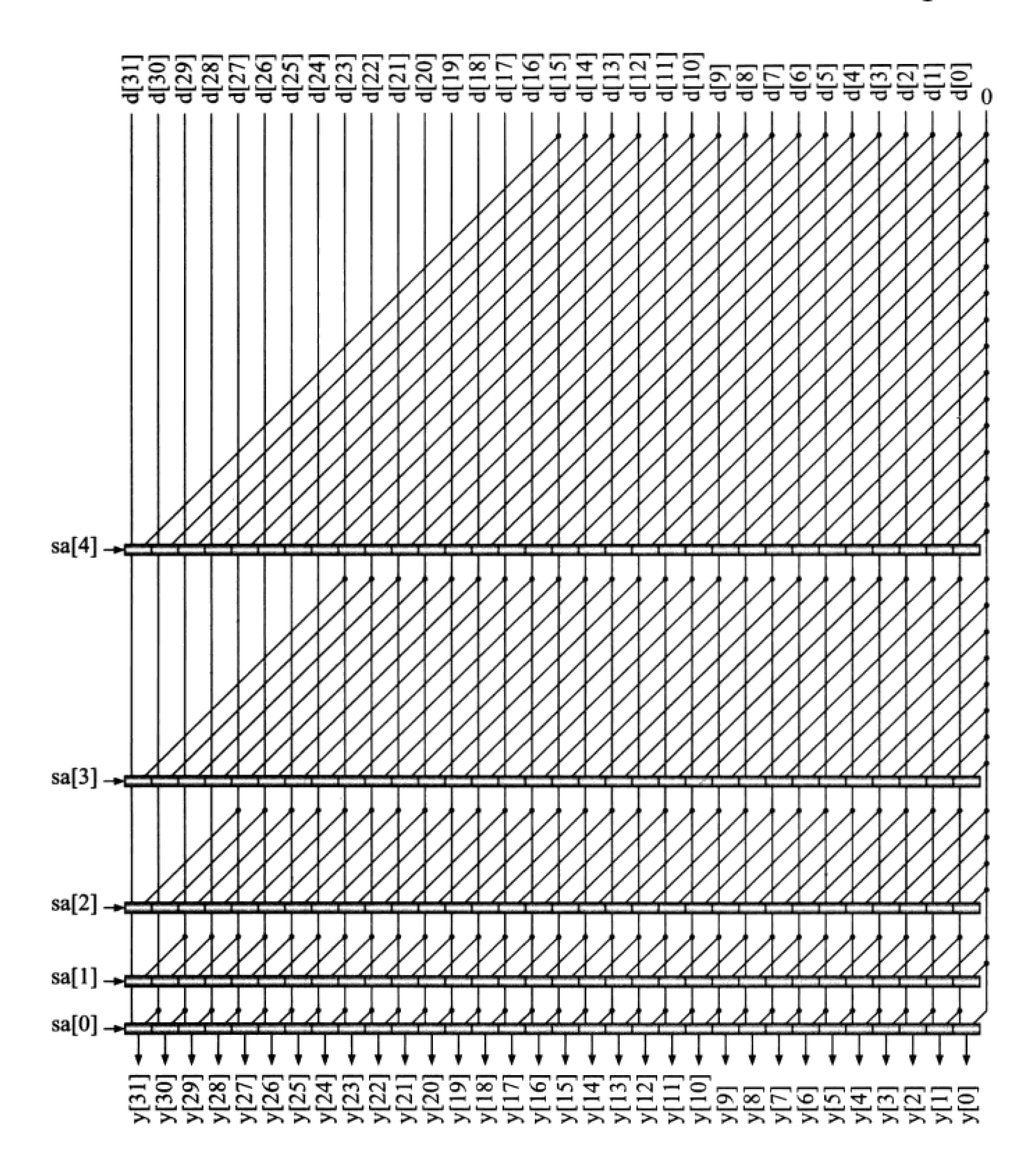


图 2.15 32 位左移移位器电路图

图 2.16 所示的是 32 位移位器的整体电路图。输入信号 right = 1 时右移，否则左移；右移时，如果 arith = 1，符号扩展，否则零扩展。图中有 5 组二选一多路器，每组两个，左边一个控制左移还是右移，右边一个控制移还是不移。符号扩展由一个与门实现：当 arith = 1 时，扩展位与数据的最高位 d[31] 相同；否则扩展位为 0。图中的器件 buf 是为了产生多个与输入信号相同的信号。

以下是实现 32 位移位器的 Verilog HDL 代码，注意 16 位扩展位 e 的产生方法。另外，我们又有一个新的关键字 parameter，使用它定义一个 16 位的常数 z。

```
module shift_mux (d,sa,right,arith,sh);
    input [31:0]  d;
    input [4:0]   sa;
```

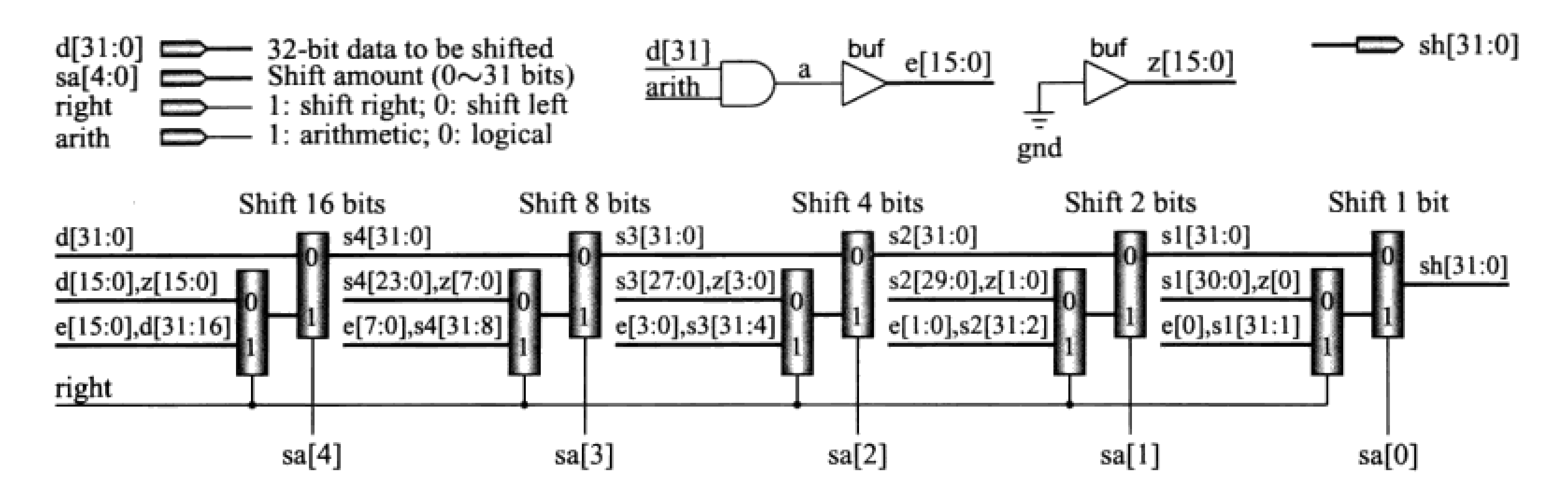


图 2.16 32 位移位器电路图

```
    input          right,arith;
    output [31:0] sh;
    wire [31:0]   t0,t1,t2,t3,t4,s1,s2,s3,s4;
    wire a = d[31] & arith;
    wire [15:0] e = {16{a}};
    parameter   z = 16'b0;
    wire [31:0] sdl4,sdr4,sdl3,sdr3,sdl2,sdr2,sdl1,sdr1,sdl0,sdr0;
    assign        sdl4 = {d[15:0],z};        // shift left  16-bit
    assign        sdr4 = {e,d[31:16]};       // shift right 16-bit
    mux2x32 m_right4 (sdl4,sdr4,right,t4);   // left or right
    mux2x32 m_shift4 (d,t4,sa[4],s4);        // not_shift or shift
    assign        sdl3 = {s4[23:0],z[7:0]}; // shift left  8-bit
    assign        sdr3 = {e[7:0],s4[31:8]}; // shift right 8-bit
    mux2x32 m_right3 (sdl3,sdr3,right,t3);   // left or right
    mux2x32 m_shift3 (s4,t3,sa[3],s3);       // not_shift or shift
    assign        sdl2 = {s3[27:0],z[3:0]}; // shift left  4-bit
    assign        sdr2 = {e[3:0],s3[31:4]}; // shift right 4-bit
    mux2x32 m_right2 (sdl2,sdr2,right,t2);   // left or right
    mux2x32 m_shift2 (s3,t2,sa[2],s2);       // not_shift or shift
    assign        sdl1 = {s2[29:0],z[1:0]}; // shift left  2-bit
    assign        sdr1 = {e[1:0],s2[31:2]}; // shift right 2-bit
    mux2x32 m_right1 (sdl1,sdr1,right,t1);   // left or right
    mux2x32 m_shift1 (s2,t1,sa[1],s1);       // not_shift or shift
    assign        sdl0 = {s1[30:0],z[0]};    // shift left  1-bit
    assign        sdr0 = {e[0],s1[31:1]};    // shift right 1-bit
    mux2x32 m_right0 (sdl0,sdr0,right,t0);   // left or right
    mux2x32 m_shift0 (s1,t0,sa[0],sh);       // not_shift or shift
endmodule
```

我们把以上的代码归为结构描述风格。测试代码和仿真结果如下所示。

```
`include "shift_mux.v"
`include "mux2x32.v"
module shift_mux_test;
```

```
    reg [31:0] d;
    reg [4:0]  sa;
    reg        right,arith;
    wire [31:0] sh;
    shift_mux sft (d,sa,right,arith,sh);
    initial begin
           right=0; arith=0; d=32'hff0000ff; sa=5'h8;
        #1 right=1; arith=0; d=32'hff0000ff; sa=5'h8;
        #1 right=1; arith=1; d=32'hff0000ff; sa=5'h8;
        #1 right=0; arith=0; d=32'hff0000ff; sa=5'h0;
        #1 $finish;
    end
    initial begin
        $monitor($time," right=%b",right," arith=%b",
                 arith," d=%h",d," sa=%h",sa," sh=%h",sh);
        $dumpfile("shift_mux.vcd");
        $dumpvars;
    end
endmodule
```

```
[yamin@localhost cpu]$ iverilog shift_mux_test.v
[yamin@localhost cpu]$ vvp a.out
VCD info: dumpfile shift_mux.vcd opened for output.
                   0 right=0 arith=0 d=ff0000ff sa=08 sh=0000ff00
                   1 right=1 arith=0 d=ff0000ff sa=08 sh=00ff0000
                   2 right=1 arith=1 d=ff0000ff sa=08 sh=ffff0000
                   3 right=0 arith=0 d=ff0000ff sa=00 sh=ff0000ff
[yamin@localhost cpu]$
```

可能有读者觉得以上代码太烦琐了。嗯，感觉很对。有简单的，下面的就是：

```
module shift (d,sa,right,arith,sh);
    input [31:0] d;
    input [4:0]  sa;
    input        right,arith;
    output [31:0] sh;
    reg [31:0]    sh;
    always @* begin
        if (!right) begin          // shift left
            sh = d << sa;
        end else if (!arith) begin // shift right logical
            sh = d >> sa;
        end else begin             // shift right arithmetic
            sh = $signed(d) >>> sa;
        end
    end
endmodule
```

这是一种功能描述风格的代码。注意 always 右面的条件，这种写法只有新的 Verilog HDL 版本才支持。3 个大于号连在一起用表示算术右移。$signed(d) 表示变量 d 是一个带符号数。看过了以上实现移位器的两个版本，你是否觉得用结构描述和数据流风格的 Verilog HDL 代码设计电路像用汇编语言编写程序，而用功能描述风格的代码设计电路像用 C 语言编写程序？请回答 “是” 。

以上讲述的内容全部属于组合电路。它们的特点是输出仅与当前输入有关。逻辑电路还有另外一种类型，而且是重要的类型，它就是时序电路。

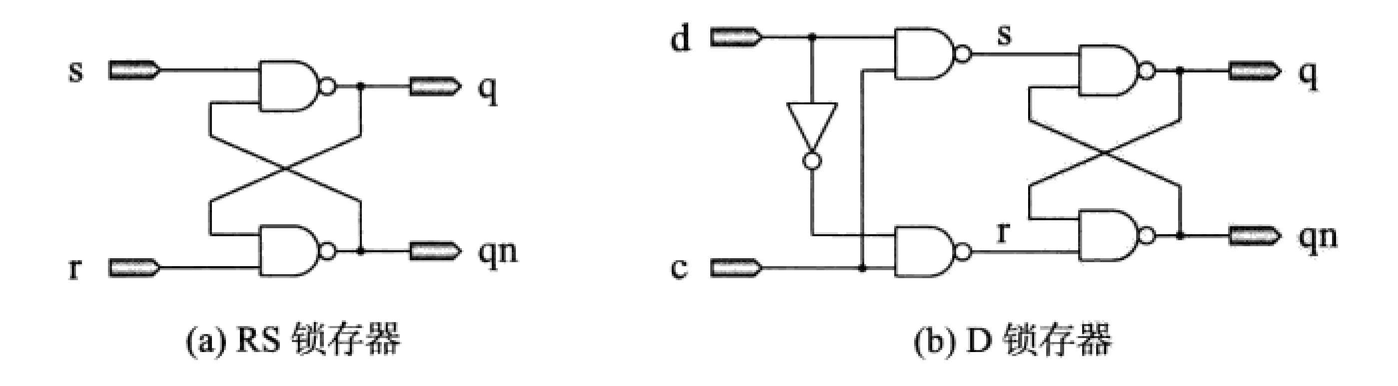
2.6 时序电路的设计方法

时序电路 (Sequential Logic Circuits) 不仅与当前输入有关，而且与 “历史” 有关。而历史与以前的输入有关。举一个例子，假如你想用一大堆硬币从自动售货机买饮料。如果自动售货机只知道你 “这次” 放入了多少钱而忘记了你已经放入的钱，那么第一，你亏了；第二，也许你买不到饮料，尽管你很渴并有很多零钱；第三，你会埋怨设计这台自动售货机的人水平不是一般的低，而是相当低；第四，不知道除了埋怨你还将做些什么。

当然不会有这种情况发生，作者只是举个例子。要想知道 “历史” ，必要的条件是会 “记忆” 。电路能记忆？能！存储器就是会记忆的器件。存储器的原理我们放在以后再讲，这里先说说两种能记忆的电路：D 锁存器和 D 触发器。

2.6.1 D 锁存器

D 锁存器 (D Latch) 是 “电平触发” 的记忆电路 (见图 2.17)。它的基本记忆单元是 “RS 锁存器” ，如图 2.17(a) 所示。R 代表 Reset，低电平有效；S 代表 Set，也是低电平有效。与组合电路相比，记忆单元的特点是信号有反馈。你试着把 RS 锁存器中下面的与非门使劲地往左拉拉看，你会发现 q 的信号又折返回来了。当 q = 0 时，它 “封锁” 了 r 输入端。由于电路是对称的，qn 也能封锁 s。而组合电路是从左至右一方通行的单行道，谁也封锁不了谁。



(a) RS 锁存器 (b) D 锁存器

图 2.17 D 锁存器电路图

当 r = 0 (有效) 且 s = 1 (无效) 时，电路 “复位” ，或称 “清零” ，输出 q 为 0，qn (可以理解为 q 的非) 为 1；当 r = 1 且 s = 0 时，电路 “置 1” ，输出 q 为 1，qn

为 0；当 r = 1 且 s = 1 时，电路保持原来的状态不变；r = 0 且 s = 0？想要干什么呢？这样干也不是不可以，只是 q 和 qn 都输出 1，意义不明确。

好在 D 锁存器 (图 2.17(b)) 不会出现这种意义不明确的举动。当 c = 1 (高电平) 时，d 被锁存，由 q 送出；当 c = 0 (低电平) 时，d 被封锁，q 的输出不受 d 的影响。注意，当 c 为高电平时，q“跟随”d 的变化而变化，这一点与 D 触发器不同。

以下是 D 锁存器的 Verilog HDL 代码。注意，以下的两个模块可放在一个文件中，文件名为 d_latch.v。在一个文件中可以写多个模块，其中有一个是主模块。文件名应使用主模块名。本例中 d_latch 是主模块，它调用 rs_latch 模块。

```
module d_latch (c,d,q,qn);
    input c,d;
    output q,qn;
    wire r,s;
    nand nand1 (s, d,c);
    nand nand2 (r,~d,c);
    rs_latch rslatch (s,r,q,qn);
endmodule

module rs_latch (s,r,q,qn);
    input s,r;
    output q,qn;
    nand nand1 (q,s,qn);
    nand nand2 (qn,r,q);
endmodule
```

图 2.18 所示的是 D 锁存器代码在 Quartus II 上的仿真结果。从图中我们看出，当 c 为高电平时，q “跟随” d 的变化而变化；c 为低电平时，q 不变。注意开始处的输出：虽然 q 不变，但 q 原来是 0 还是 1 并不知道。还是波形图看起来比较直观。由于有了 Quartus II，作者就不怎么用 gtkwave 了。其中的一个主要原因是使用 Quartus II 时不用写测试代码了，直接编辑输入信号的波形就行。

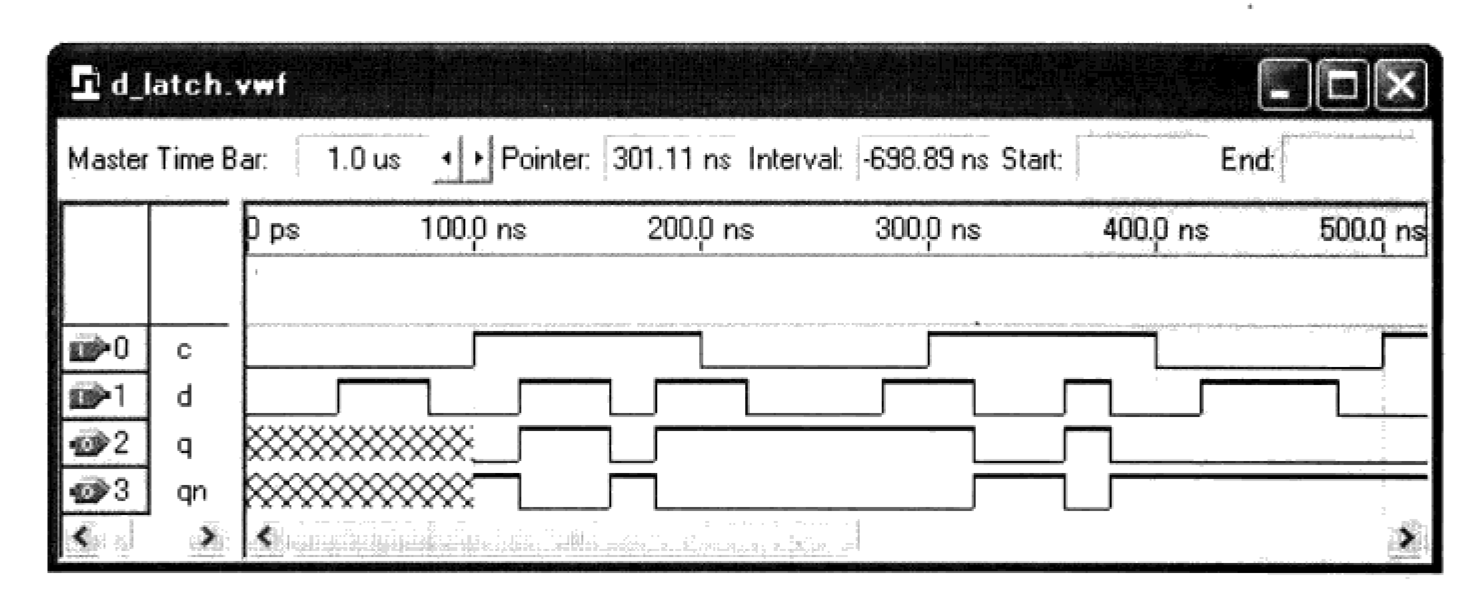


图 2.18 D 锁存器仿真结果

2.6.2 D 触发器

D 触发器 (D Flip Flop) 是“时钟上升沿触发”的记忆电路，不会有“跟随”现象发生。D 触发器有“学术界”和“工业界”两个版本，以下分别介绍。

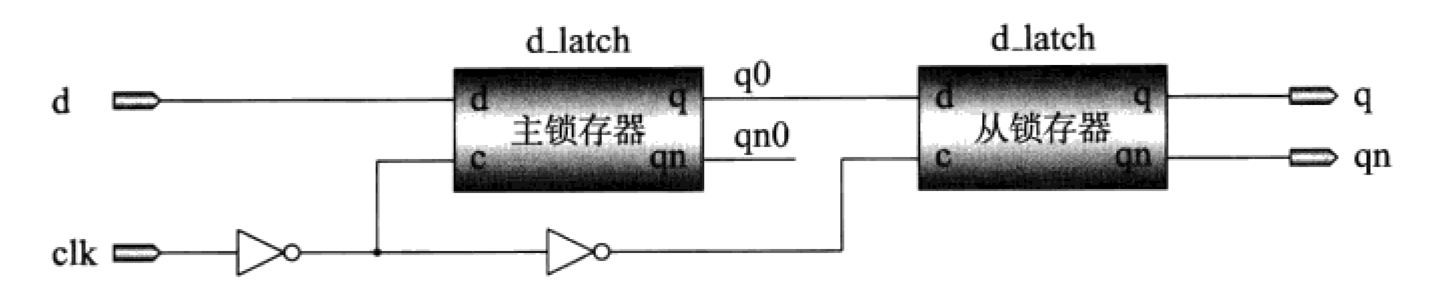


图 2.19 学术界版本的 D 触发器电路图

先讲学术界的版本，见图 2.19。很简单，使用两个非门和两个 D 锁存器，左边的叫“主锁存器”，右边的叫“从锁存器”。时钟 clk 的低电平 (经过一个非门变成高电平) 把 d 的数据“锁存”进主锁存器，输出为 q0；时钟的高电平再把 q0 “锁存”进从锁存器。这样，主锁存器在时钟的低电平期间对 d 的跟随就不会影响到最终的输出 q。虽然在时钟的高电平期间 q 跟随 q0，但这时的 q0 不会发生变化了。因为输出信号 q 的变化发生在时钟信号电平从低到高的瞬间，所以 D 触发器是“时钟上升沿触发”。该电路的 Verilog HDL 代码如下。它调用了两个 D 锁存器 d_latch。

```
module d_flip_flop (clk,d,q,qn);
    input clk,d;
    output q,qn;
    wire   q0,qn0;
    d_latch dlatch1 (~clk,d,q0,qn0);
    d_latch dlatch2 ( clk,q0,q,qn);
endmodule
```

仿真结果如图 2.20 所示。输入信号的波形与图 2.18 所示的对 D 锁存器进行仿真时的输入信号的波形完全相同，但二者的输出大不相同。

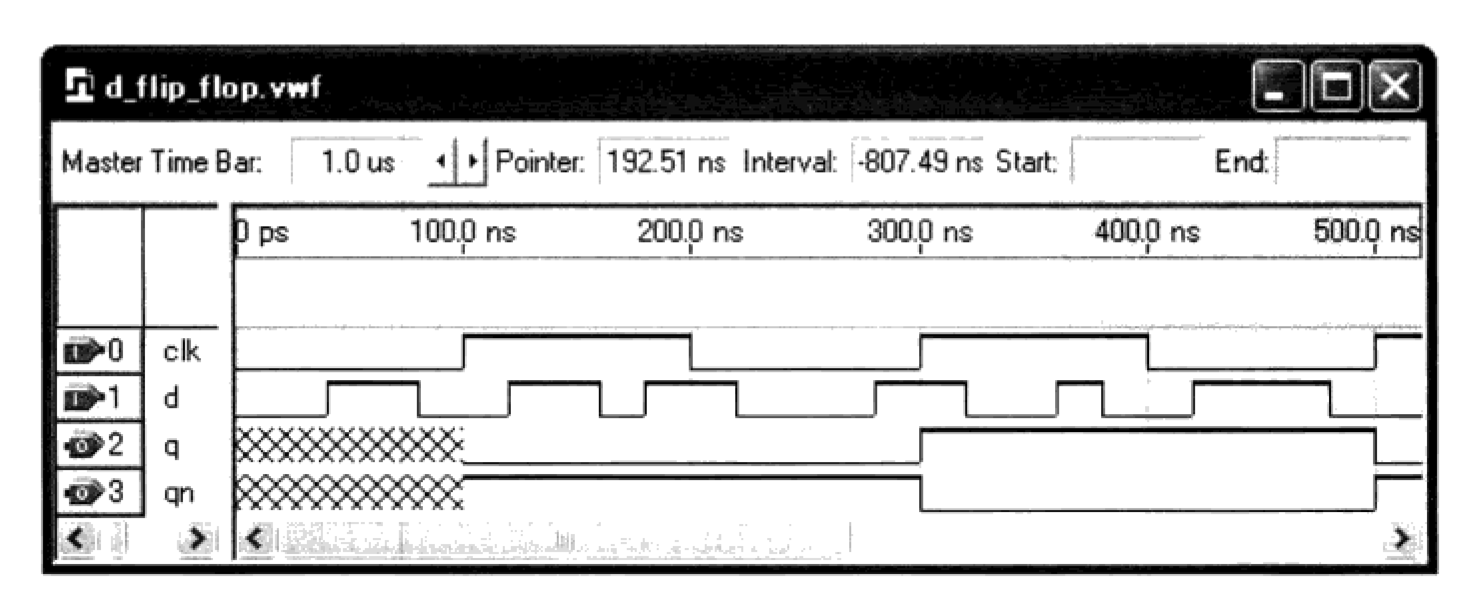


图 2.20 学术界版本的 D 触发器仿真结果

工业界版本的 D 触发器如图 2.21 所示，其中的 prn 和 clrn 分别为低电平有效的置 1 和清零端。仿真结果如图 2.22 所示，注意看时钟 (clk) 上升沿处 Q 的输出。

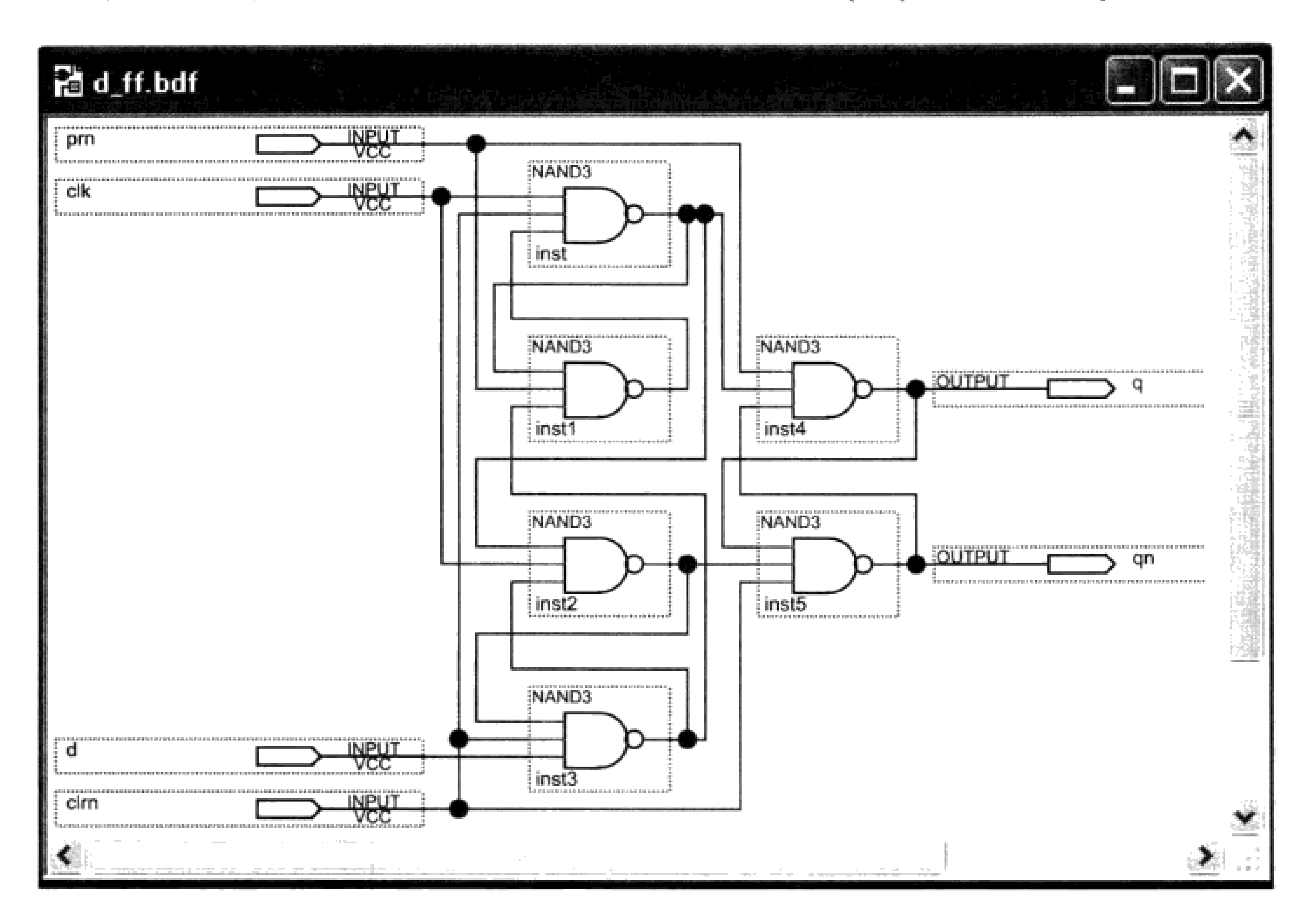


图 2.21 工业界版本的 D 触发器电路图

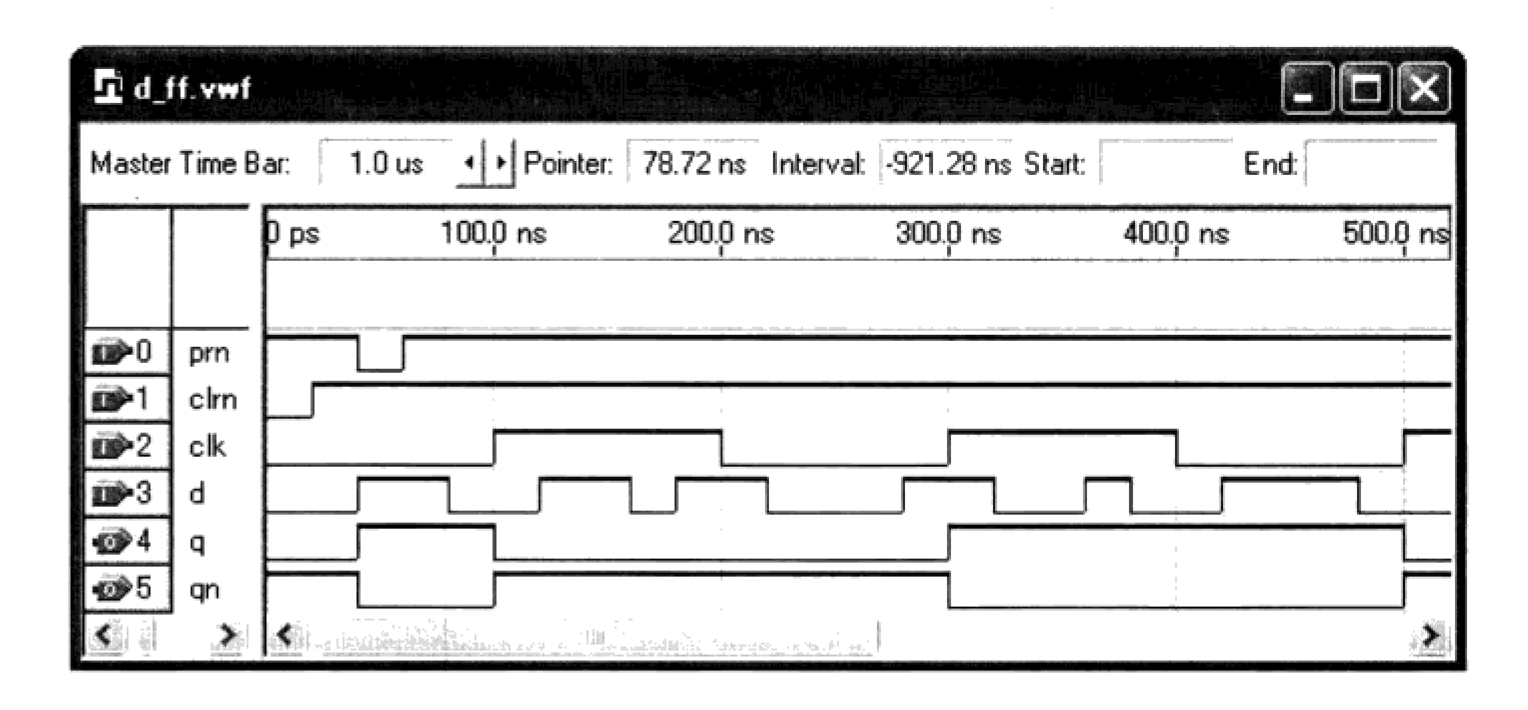


图 2.22 工业界版本的 D 触发器仿真结果

D 触发器可以用功能描述风格的 Verilog HDL 来设计。以下的两个版本分别是带有“同步”清零端的 D 触发器和带有“异步”清零端的 D 触发器。同步是指在时钟上升沿处检查清零信号是否有效；异步是指清零端与时钟无关，只要有效就清零。

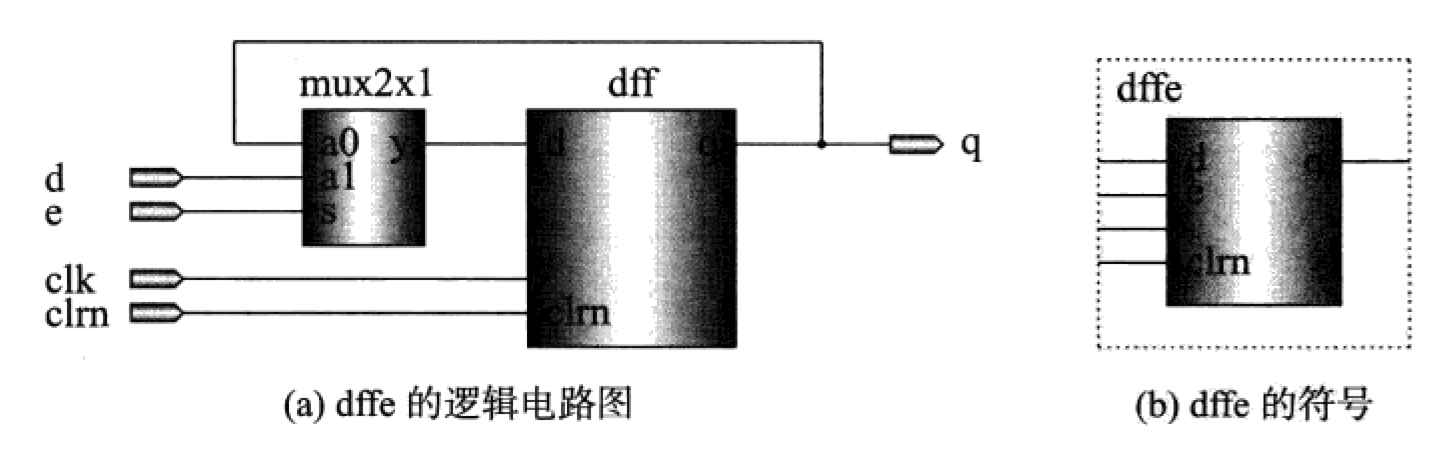
带有同步清零端的 D 触发器 Verilog HDL 代码如下，其中 posedge 是关键字，代表上升沿。

```
module d_ff (d,clk,clrn,q); // dff with synchronous reset.
    input d,clk,clrn;
    output q;
    reg    q;
    always @ (posedge clk) begin
        if (clrn == 0) q <= 0;
        else           q <= d;
    end
endmodule
```

带有异步清零端的 D 触发器的 Verilog HDL 代码如下，其中 negedge 是关键字，代表下降沿。

```
module dff (d,clk,clrn,q); // dff with asynchronous reset.
    input d,clk,clrn;
    output q;
    reg    q;
    always @ (posedge clk or negedge clrn) begin
        if (clrn == 0) q <= 0;
        else           q <= d;
    end
endmodule
```

图 2.23 所示的是带有使能端 e (Enable) 的 D 触发器 dffe。我们使用了一个二选一多路器。当 e = 1 时，它与普通的 D 触发器相同；当 e = 0 时，禁止新的数据的写入，保持原来的状态不变。实际上是把原来 q 的数据经多路器又往 D 触发器中写了一次。有人把 e 端与输入信号 clk 相与后再接到 D 触发器的时钟端。这是一种很不好的设计，由于 e 由其他电路产生，可能会有“毛刺”，导致向 D 触发器中误写入。还有就是由于 e 的原因造成上升沿不同程度的延迟，可能导致同步电路锁存错误。



(a) dffe 的逻辑电路图　　(b) dffe 的符号

图 2.23　带有使能端的 D 触发器 dffe 电路图

异步清零再加上使能端 e 的 D 触发器 (dffe) 的 Verilog HDL 代码如下。

```
module dffe (d,clk,clrn,e,q);
    input d;
    input clk,clrn,e;
    output q;
    reg   q;
    always @(negedge clrn or posedge clk)
        if (clrn == 0) q <= 0;
        else if (e)    q <= d;
endmodule
```

2.6.3 状态转移图及时序电路设计

以上介绍了 D 锁存器和 D 触发器，它们只是单纯的记忆器件。现在我们通过一个例子说明如何设计时序电路。

假设我们要设计一个六进制的计数器。我们有一个输入信号 u。当 u = 1 时，计数次序为 0, 1, 2, 3, 4, 5, 0, 1, 2, · · ·；当 u = 0 时，计数次序为 5, 4, 3, 2, 1, 0, 5, 4, 3, · · ·。另外，计数器的值要用七段显示器显示出来，见图 2.24。

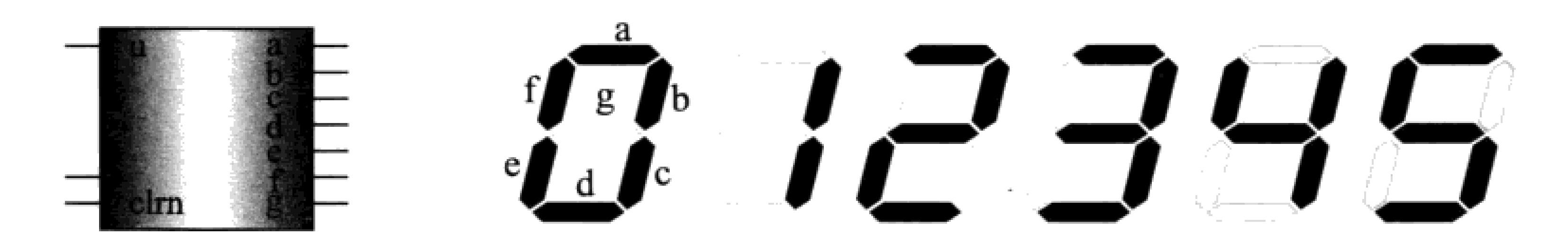


图 2.24 六进制的计数器符号及七段显示器

很明显，计数器有 6 个状态。我们要用 D 触发器来表示或区分出这 6 个状态。问题是要用多少个 D 触发器才够。6 个？不需要那么多，3 个就行。由于 3 个 D 触发器能够存储 3 位二进制数，而 3 位二进制数能表示 $2^3 = 8$ 个状态，它们分别是 000, 001, 010, 011, 100, 101, 110, 111。我们只有 6 个状态，还富余两个状态。一般来讲，N 个状态需要 $\lceil \log_2 N \rceil$ 个 D 触发器，其中 $\lceil\ \rceil$ 是向上取整数的意思。

图 2.25 所示是六进制计数器的总体电路图，其中左半部分是 3 个 D 触发器，用于记录计数器当前的状态。右半部分是组合逻辑，生成下一状态信号并产生七段显示器的控制信号。

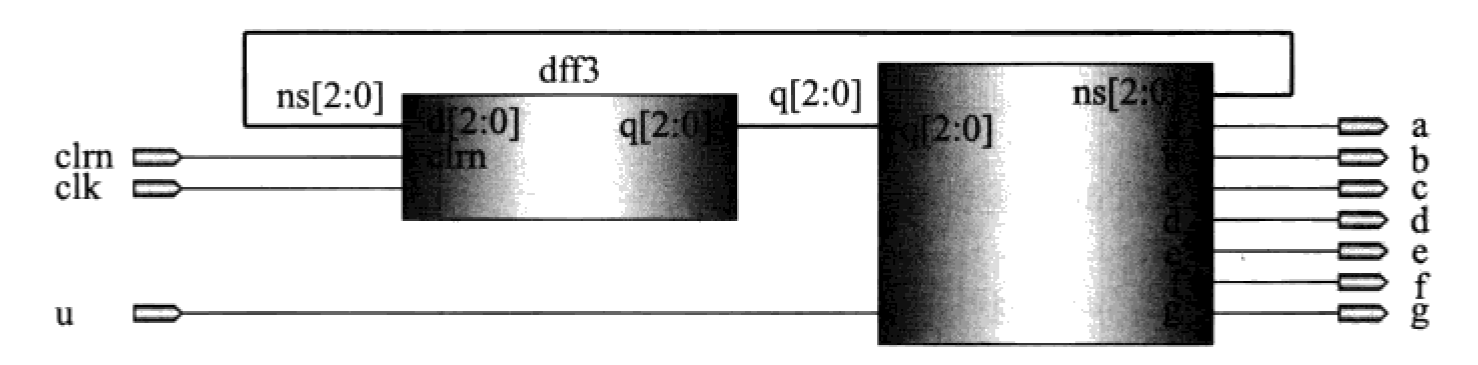


图 2.25 六进制计数器的总体电路图

现在，我们要画出“状态转移图”。首先给每个状态起一个名字，假设我们分别用 S0 ～ S5 表示计数器的值 0 ～ 5。则我们可以画出如图 2.26 所示的状态转移图。

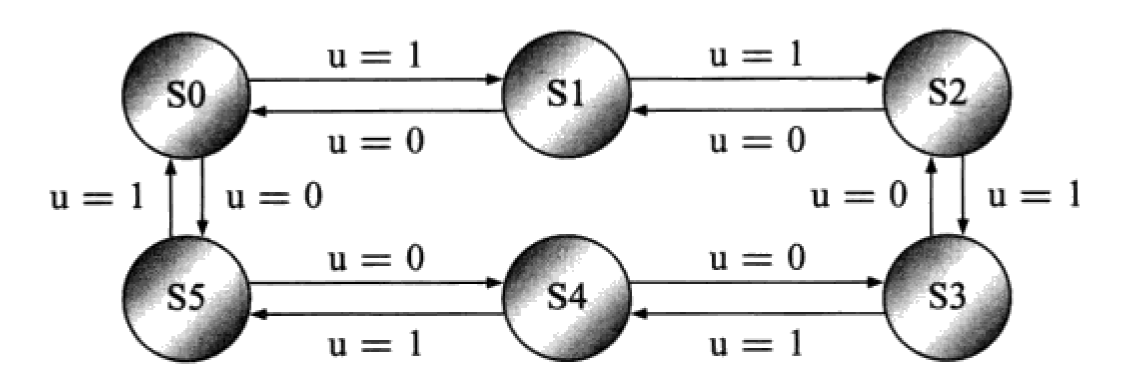


图 2.26 六进制计数器的状态转移图

有了状态转移图，我们可以制作一个状态转移表。在制表之前，我们先给每个状态赋一个 3 位二进制的值。因为 3 位二进制数总共有 8 个不同的值，我们可以从中任选 6 个，分别赋给 6 个状态。注意，给状态赋值可以任意，只要保证每个状态的值是唯一的就行了。表 2.4 所示的仅是一种可能的赋值方案。

表 2.4 给 6 个状态赋不同 3 位二进制数值的一种方案

状态	S0	S1	S2	S3	S4	S5
二进制值	000	001	010	011	100	101

D 触发器的输出信号是“当前状态”，用 q[2..0] 表示它。在时钟的上升沿处，“下一状态”要被写入到 D 触发器中。我们用 ns[2..0] 表示下一状态。表 2.5 是六进制计数器的状态转移表，通过它求出表示下一状态的 ns[2..0] 每一位的逻辑表达式。

表 2.5 六进制计数器的状态转移表

当前状态				输入	下一状态			
	q[2]	q[1]	q[0]	u		ns[2]	ns[1]	ns[0]
S0	0	0	0	1	S1	0	0	1
				0	S5	1	0	1
S1	0	0	1	1	S2	0	1	0
				0	S0	0	0	0
S2	0	1	0	1	S3	0	1	1
				0	S1	0	0	1
S3	0	1	1	1	S4	1	0	0
				0	S2	0	1	0
S4	1	0	0	1	S5	1	0	1
				0	S3	0	1	1
S5	1	0	1	1	S0	0	0	0
				0	S4	1	0	0

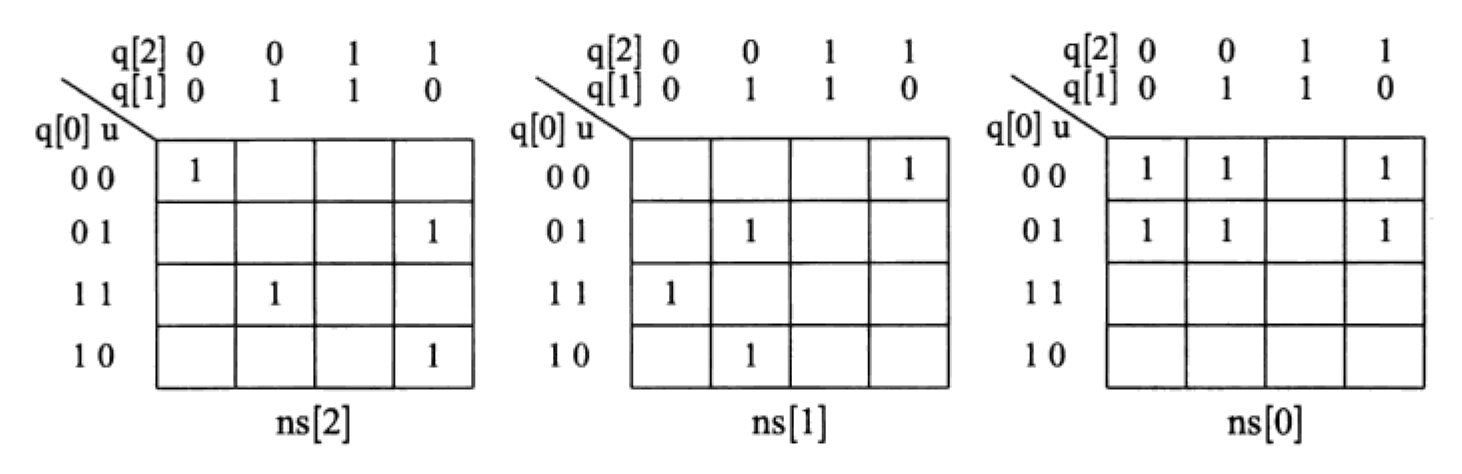


图 2.27　下一状态变量的卡诺图化简

求下一状态的每一位的逻辑表达式，可以使用如图 2.27 所示的卡诺图。从图中我们看出只有 ns[0] 可以化简。我们写出 ns 每一位的表达式如下。

$$\mathrm{ns}[0] = \overline{q[2]}\,\overline{q[0]} + \overline{q[1]}\,\overline{q[0]}$$

$$\mathrm{ns}[1] = \overline{q[2]}\,\overline{q[1]}\,q[0]\,u + \overline{q[2]}\,q[1]\,\overline{q[0]}\,u + \overline{q[2]}\,q[1]\,q[0]\,\overline{u} + q[2]\,\overline{q[1]}\,\overline{q[0]}\,\overline{u}$$

$$\mathrm{ns}[2] = \overline{q[2]}\,\overline{q[1]}\,\overline{q[0]}\,\overline{u} + \overline{q[2]}\,q[1]\,q[0]\,u + q[2]\,\overline{q[1]}\,\overline{q[0]}\,u + q[2]\,\overline{q[1]}\,q[0]\,\overline{u}$$

q[2 : 0]	g f e d c b a
0 0 0	1 0 0 0 0 0 0
0 0 1	1 1 1 1 0 0 1
0 1 0	0 1 0 0 1 0 0
0 1 1	0 1 1 0 0 0 0
1 0 0	0 0 1 1 0 0 1
1 0 1	0 0 1 0 0 1 0

a

q[0] \ q[2] q[1]	0 0	0 1	1 1	1 0
0			x	1
1	1		x	

b

q[0] \ q[2] q[1]	0 0	0 1	1 1	1 0
0			x	
1			x	1

c

q[0] \ q[2] q[1]	0 0	0 1	1 1	1 0
0		1	x	
1			x	

d

q[0] \ q[2] q[1]	0 0	0 1	1 1	1 0
0			x	1
1	1		x	

e

q[0] \ q[2] q[1]	0 0	0 1	1 1	1 0
0			x	1
1	1	1	x	1

f

q[0] \ q[2] q[1]	0 0	0 1	1 1	1 0
0		1	x	
1	1	1	x	

g

q[0] \ q[2] q[1]	0 0	0 1	1 1	1 0
0	1		x	
1	1		x	

图 2.28　输出信号的卡诺图化简

同样，我们可以用卡诺图化简七段显示器的 7 个控制信号，如图 2.28 所示 (假设低电平点亮 LED)。我们得到如下的输出信号的逻辑表达式。

$$a = \overline{q[2]}\,\overline{q[1]}\,q[0] + q[2]\,\overline{q[0]}$$

$$b = q[2]\,q[0]$$

$$c = q[1]\,\overline{q[0]}$$

$$d = a$$

$$e = q[2]\,\overline{q[0]} + q[0]$$

$$f = q[1]\,\overline{q[0]} + \overline{q[2]}\,q[0]$$

$$g = \overline{q[2]}\ \overline{q[1]}$$

图 2.29 ～ 图 2.32 是六进制计数器的详细电路，图 2.33 是仿真结果。

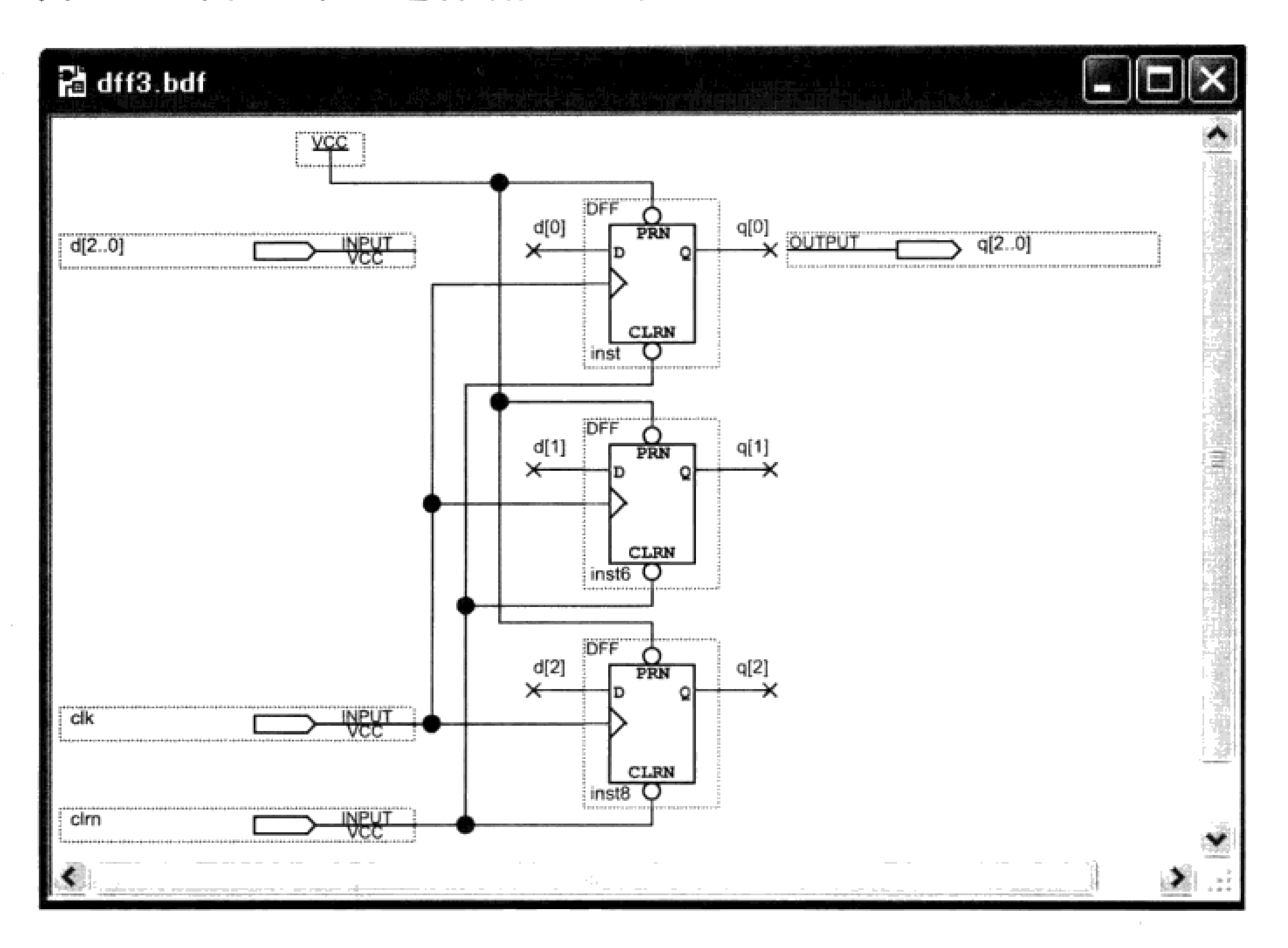


图 2.29　六进制计数器中的 3 位状态寄存器电路图

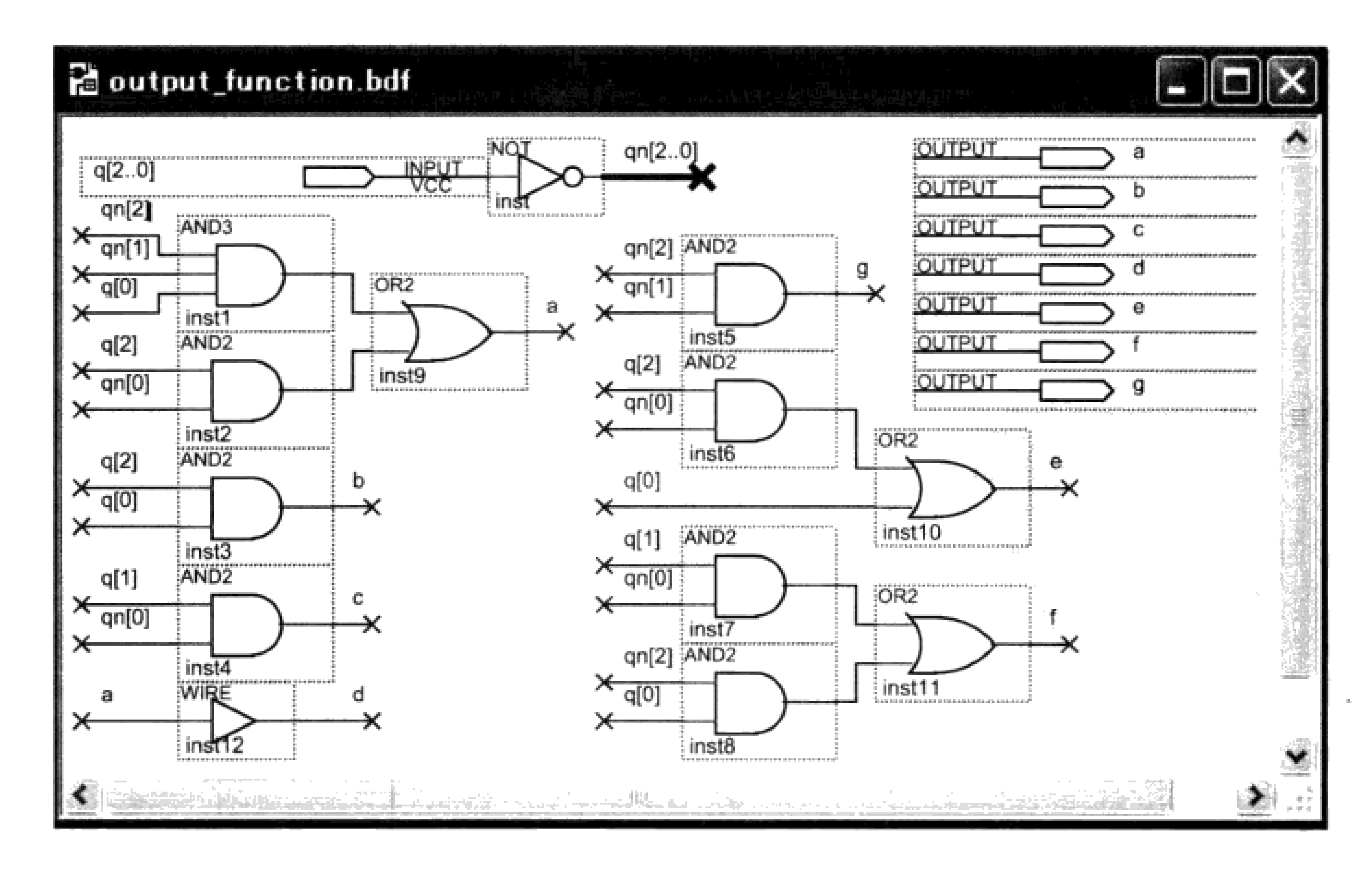


图 2.30　六进制计数器中的输出信号产生电路

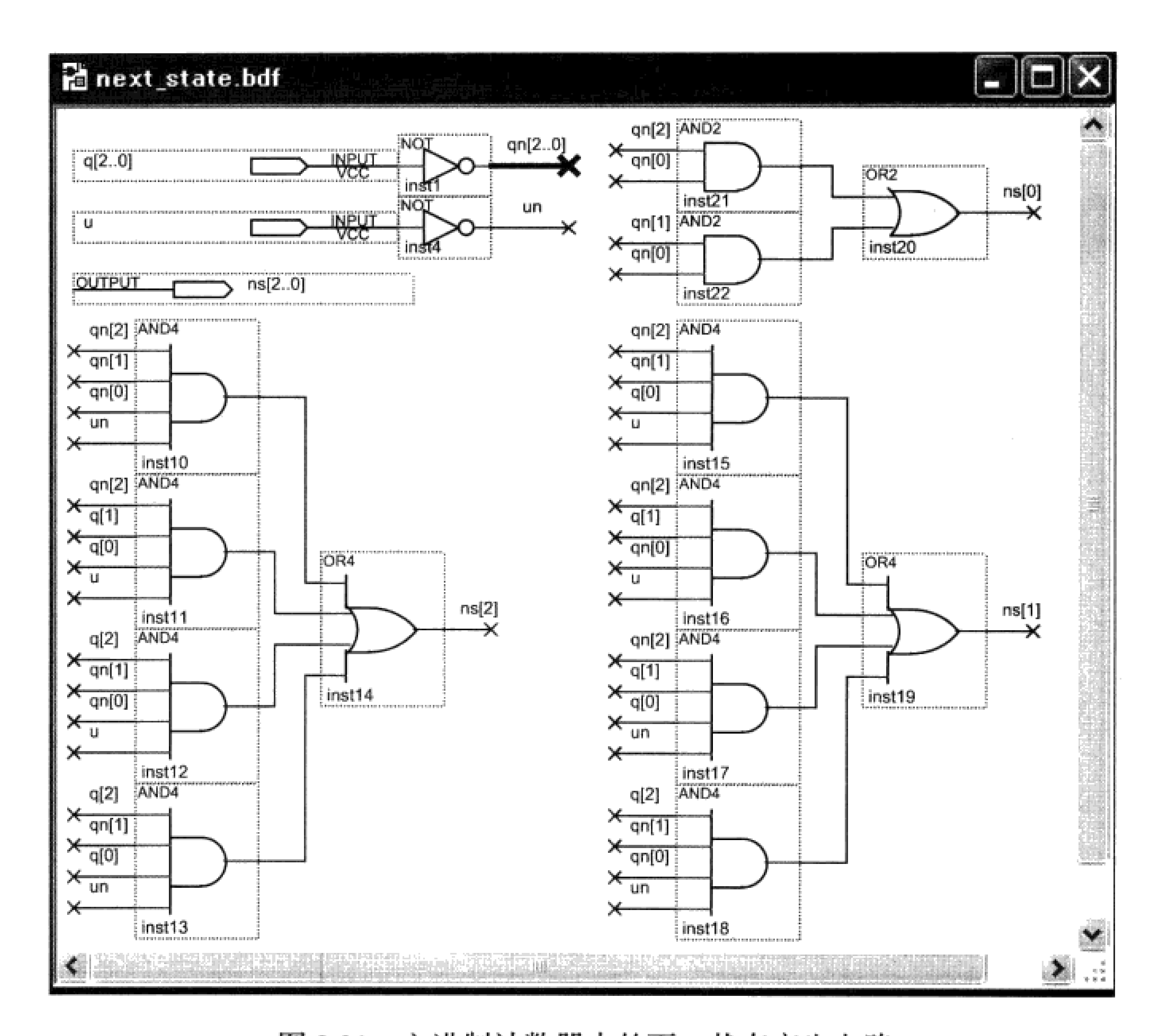


图 2.31 六进制计数器中的下一状态产生电路

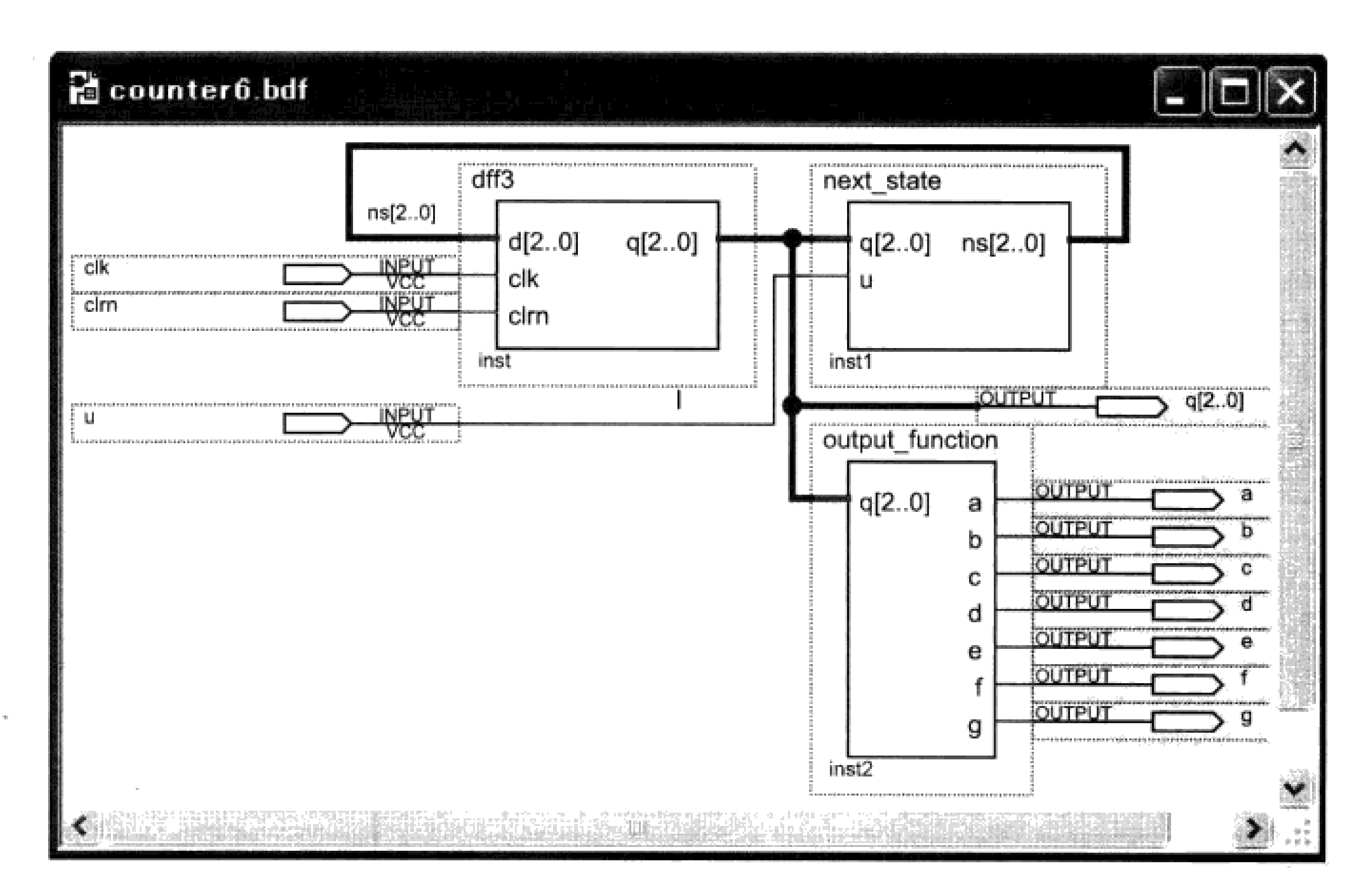


图 2.32 六进制计数器的总体电路

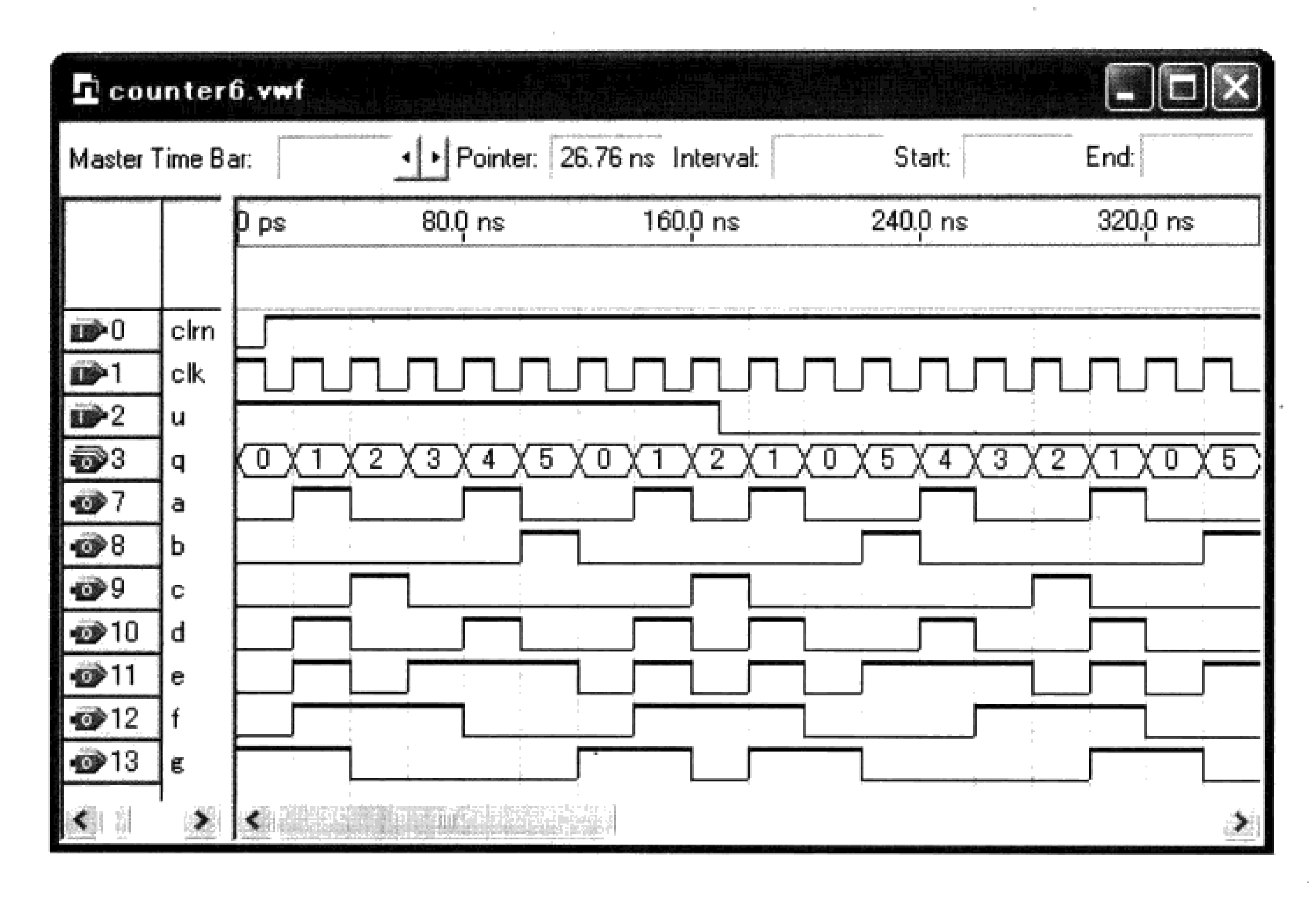


图 2.33 六进制计数器的仿真结果

我们讲述计数器设计的详细过程的目的不是仅为了设计一个计数器，而是让读者掌握设计时序电路的方法。其实，如果只想设计一个带有七段显示的六进制的计数器，很简单，看下面的 counter6.v。

```
module counter6 (u,clk,clrn,q,a,b,c,d,e,f,g);
    input u,clk,clrn;
    output [2:0] q;
    output       a,b,c,d,e,f,g;

    reg [2:0]    q;
    always @ (posedge clk or negedge clrn) begin
        if (clrn == 0) q <= 0;
        else if (u) q <= (q + 1) % 6;
             else if (q == 0) q <= 5;
                  else q <= (q - 1) % 6;
    end

    assign {g,f,e,d,c,b,a} = seg7(q);
    function [6:0] seg7;
        input [2:0] q;
        case (q)
            3'd0 : seg7 = 7'b1000000;
            3'd1 : seg7 = 7'b1111001;
            3'd2 : seg7 = 7'b0100100;
```

```
            3'd3 : seg7 = 7'b0110000;
            3'd4 : seg7 = 7'b0011001;
            3'd5 : seg7 = 7'b0010010;
            default: seg7 = 7'b1111111; // light off
        endcase
    endfunction
endmodule
```

一般的时序电路的结构如图 2.34 所示。时序电路的实现方式有 Mealy Model 和 Moore Model。它们之间的区别看图就知道了。有关时序电路设计的内容，在介绍多周期 CPU 的 Verilog HDL 实现方法时，我们还要详细讲解。

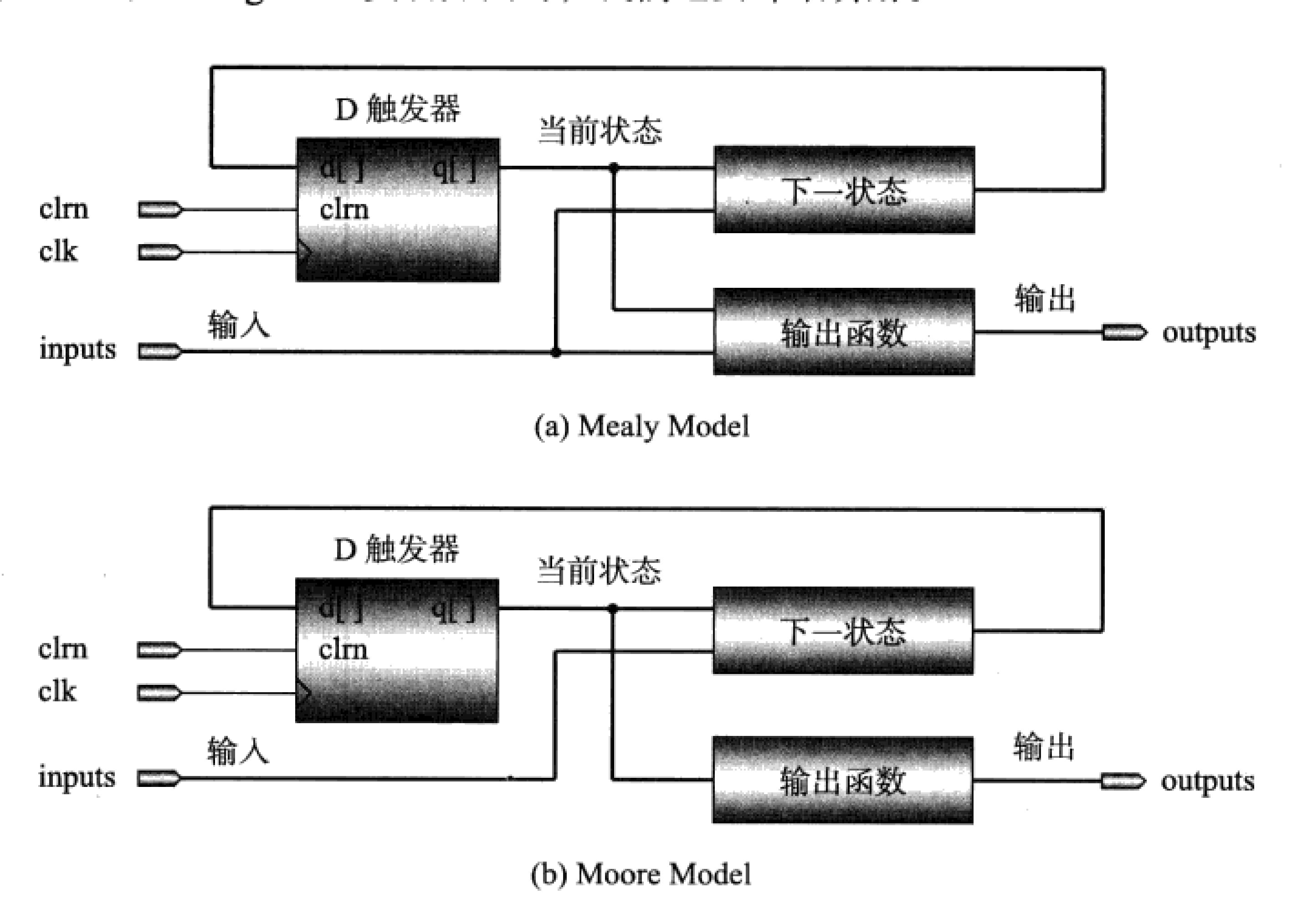


图 2.34　时序电路示意图

2.7　习题

1. 下载并安装 Icarus Verilog (或其他的 Verilog HDL 编译及仿真器)。
2. 参照图 2.10(a)，试写出 cmoscmos.v 的 Verilog HDL 代码。
3. 试设计一个 3 输入的 CMOS 与非门和一个 3 输入的 CMOS 或非门。
4. 试只使用或非门设计一个一位二选一多路器。
5. 试用多个 32 位二选一多路器 mux2x32 设计一个 32 位的八选一多路器 mux8x32。
6. 试分别用结构描述、数据流描述和功能描述三种风格写出带有使能端的 3-8 译码器的 Verilog HDL 代码。提示：功能描述风格的语句可用 `e = ena << n`。

7. 在中断控制电路中通常要用到优先级编码器 (Priority Encoder)，指出是否有中断以及中断的优先级 (向量)。设计一个 8-3 优先级编码器。提示：可使用 casex 语句。不知道什么是优先级编码器和 casex？到网上去查。
8. 你想只用或非门设计一个 D 锁存器吗？如果想，试试看。
9. 用 Verilog HDL 设计一个异步清零再加上使能端 e 的 32 位 D 触发器 (dffe32)。
10. 试读懂下面的代码，说明它的功能并在 FPGA 板上运行。

```
module minute_second (clk, m1, m0, s1, s0, dots);
    input  clk; // 50MHz
    output [6:0] m1, m0; // munite 7-seg
    output [6:0] s1, s0; // second 7-seg
    output [3:0] dots;   // 4 decimal points

    reg sec_clk = 1; // second clock
    reg [24:0] clk_cnt = 0;
    always @ (posedge clk) begin
        if (clk_cnt == 25'd24999999) begin
            clk_cnt <= 0;
            sec_clk <= ~sec_clk;
        end else begin
            clk_cnt <= clk_cnt + 25'd1;
        end
    end

    reg [2:0] min1 = 0, sec1 = 0;
    reg [3:0] min0 = 0, sec0 = 0;
    always @ (posedge sec_clk) begin
        if (sec0 == 4'd9) begin
            sec0 <= 0;
            if (sec1 == 3'd5) begin
                sec1 <= 0;
                if (min0 == 4'd9) begin
                    min0 <= 0;
                    if (min1 == 3'd5) begin
                        min1 <= 0;
                    end else begin
                        min1 <= min1 + 3'd1;
                    end
                end else begin
                    min0 <= min0 + 4'd1;
                end
            end else begin
                sec1 <= sec1 + 3'd1;
            end
        end else begin
```

```
            sec0 <= sec0 + 4'd1;
        end
    end

    //   0
    // 5   1
    //   6
    // 4   2
    //   3
    function [6:0] seg7;
        input [3:0] q;
        case (q)
            4'd0 : seg7 = 7'b1000000;
            4'd1 : seg7 = 7'b1111001;
            4'd2 : seg7 = 7'b0100100;
            4'd3 : seg7 = 7'b0110000;
            4'd4 : seg7 = 7'b0011001;
            4'd5 : seg7 = 7'b0010010;
            4'd6 : seg7 = 7'b0000010;
            4'd7 : seg7 = 7'b1111000;
            4'd8 : seg7 = 7'b0000000;
            4'd9 : seg7 = 7'b0010000;
            default: seg7 = 7'b1111111; // light off
        endcase
    endfunction

    assign s0 = seg7(sec0);
    assign s1 = seg7({1'b0,sec1});
    assign m0 = seg7(min0);
    assign m1 = seg7({1'b0,min1});
    assign dots = 4'b1011;
endmodule
```

11. 试设计一个自动售货机的时序控制电路，条件自己想、自己设。

第 3 章　计算机算法及其 Verilog HDL 实现

本章讨论二进制整数及其加减乘除和开方运算的一般算法以及高速算法，并给出所有算法的 Verilog HDL 代码。这些算法是我们设计 CPU 时必须要用到的。

3.1　二进制整数

人类已经非常习惯了使用十进制数，对十进制数的十个符号、即阿拉伯数字 0123456789 熟悉得不能再熟悉了。但为什么是十进制而不是其他进制？是因为人类有十个手指头？不管怎样，发明这十个符号的人是很伟大的。

计算机中所有的信息，包括指令和数据，都是用二进制数表示的。一位二进制数只有两种可能的值：0 和 1 (注意这里的 0 和 1 借用了阿拉伯数字中的两个符号)。比如 1010 就是一个 4 位二进制数。

首先提一个问题：假设我们有一个 32 位的二进制数

00110011110111100000000100000000

它到底表示的是什么？正确答案是“不知道”。为什么呢？因为该二进制数的具体含义取决于它在什么场合下被使用。如果它被 MIPS CPU 当做指令来执行，则它是一条立即数加法指令 addi r30, r30, 256。为什么它是这条指令以及该指令完成什么操作等问题，我们将在第 4 章中讲述。如果它被 CPU 当做一个整数来运算，则它的整数值的大小等于 870 187 264。如果它被浮点部件 FPU 当做一个浮点数来运算，则它的值的大小等于 $+2^{-24} \times (1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-15}) =$ $+0.000000103378624771721661090850830078125$。当然，它也可能是图像或音乐中的数据，或者互联网的 IP 地址，或者其他别的什么的。

不知大家有没有发现 32 位二进制数写起来太长、太不方便了？解决这个问题的办法是用十六进制数来表示。我们把二进制数 4 位一组 4 位一组地分开，每一组用一个字符来表示。因为 4 位二进制数有 $2^4 = 16$ 种组合，所以要使用 16 个不同的符号来区分它们。这 16 个符号为 0 ～ 9 和 a ～ f，它们与 4 位二进制的 16 种组合的对应关系在表 3.1 中给出。

表 3.1　十六进制数与二进制数的对应关系

二进制	十六进制	二进制	十六进制	二进制	十六进制	二进制	十六进制
0000	0	0100	4	1000	8	1100	c
0001	1	0101	5	1001	9	1101	d
0010	2	0110	6	1010	a	1110	e
0011	3	0111	7	1011	b	1111	f

有了十六进制的表示方法，上面的 32 位二进制数可以写成

$$0011_0011_1101_1110_0000_0001_0000_0000_2 = 33de0100_{16}$$

十六进制数与二进制数之间并没有本质的区别，只是表示方法不同而已。如果要把二进制数用十进制数来表示，问题就没那么简单了。问题不简单的意思有两个，一个是要确定该二进制数是什么表示方法，另一个是转换起来也比较麻烦。

以下我们只介绍两种常用的二进制整数：(1) 无符号二进制整数，也就是绝对值；(2) 带符号二进制整数，能表示正数，也能表示负数。

3.1.1 无符号二进制整数

设有 n 位二进制数 $b_{n-1}b_{n-2}\cdots b_1b_0$，其中 b_i $(i = 0, 1, \cdots, n-2, n-1)$ 的值为 0 或 1。如果我们用它来表一个无符号数，它的值为

$$b_{n-1}b_{n-2}\cdots b_1\, b_0 = b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + \cdots + b_1\times 2^1 + b_0\times 2^0$$

图 3.1 所示的是 $n = 16$ 的例子。

2^{15}	2^{14}	2^{13}	$.2^{12}$	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

图 3.1 16 位无符号数

当 n 位二进制数的所有位均为 1 时，其值为 $\sum_{i=0}^{n-1} 2^i = 2^n - 1$。这是 n 位二进制数所能表示的最大值。例如 16 位二进制数能表示的最大的无符号数为 65 535。笔者建议大家至少记住 $2^0 \sim 2^{16}$ 的十进制值。以下是 32 位无符号数所对应的十进制值：

```
0000 0000 0000 0000 0000 0000 0000 0000 = 0₁₀
0000 0000 0000 0000 0000 0000 0000 0001 = 1₁₀
0000 0000 0000 0000 0000 0000 0000 0010 = 2₁₀
··· ···
1111 1111 1111 1111 1111 1111 1111 1101 = 4 294 967 293₁₀
1111 1111 1111 1111 1111 1111 1111 1110 = 4 294 967 294₁₀
1111 1111 1111 1111 1111 1111 1111 1111 = 4 294 967 295₁₀
```

3.1.2 补码表示的带符号二进制整数

如果我们有一个负数，如何用二进制数来表示它？负数有很多种表示方法，以下 3 种是比较常用的方法。

1) 符号-绝对值：最高位是符号位 (0 表示正、1 表示负)，其余各位为绝对值。
2) 移码：从绝对值中减去一个固定的值。例如 8 位时减 127、11 位时减 1023。
3) 补码：最高位为负值，其余各位为正值 (与无符号数计算方法相同)。

本小节重点介绍补码表示方法。设有 n 位二进制数 $b_{n-1}b_{n-2}\cdots b_1b_0$，其中 b_i $(i=0,1,\cdots,n-2,n-1)$ 的值为 0 或 1。补码表示的带符号数的值为

$$b_{n-1}b_{n-2}\cdots b_1\,b_0=-b_{n-1}\times 2^{n-1}+b_{n-2}\times 2^{n-2}+\cdots+b_1\times 2^1+b_0\times 2^0$$

注意上式中等号右边 b_{n-1} 前面的负号。图 3.2 所示的是 $n=16$ 的例子。

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
$-b_{15}$	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

图 3.2 16 位补码表示的带符号数

如果最高位为 1，该二进制数一定是负数：不管其余各位如何“正”，也不会大过最高位的绝对值。以下是 32 位补码表示的带符号数所对应的十进制值：

0000 0000 0000 0000 0000 0000 0000 0000 = 0_{10}
0000 0000 0000 0000 0000 0000 0000 0001 = 1_{10}
··· ···
0111 1111 1111 1111 1111 1111 1111 1110 = $2\,147\,483\,646_{10}$
0111 1111 1111 1111 1111 1111 1111 1111 = $2\,147\,483\,647_{10}$
1000 0000 0000 0000 0000 0000 0000 0000 = $-2\,147\,483\,648_{10}$
1000 0000 0000 0000 0000 0000 0000 0001 = $-2\,147\,483\,647_{10}$
··· ···
1111 1111 1111 1111 1111 1111 1111 1110 = -2_{10}
1111 1111 1111 1111 1111 1111 1111 1111 = -1_{10}

补码表示方法所能表示的数值为 $-2^{n-1}\sim 2^{n-1}-1$，其中 n 为二进制位数。例如，16 位二进制数可以表示 $-32\,768\sim+32\,767$。表 3.2 列出了 4 位二进制数在不同的表示方法下的十进制数值。你看，同是一个二进制数，在不同的表示方法下，有不同的意义。补码表示方法有一个好处：做加减法运算时与无符号数相同。这一点我们将在 3.2 节讨论。另外，补码的全称是“2 的补码”(2’s Complement)。为什么这样称呼，我们也在 3.2 节讨论。

3.2 加减法算法及 Verilog HDL 实现

本节讨论二进制数的加减法电路的 Verilog HDL 实现以及补码表示的带符号数与无符号数之间的关系。先行进位的高速加法器的设计也在本节给出。

3.2.1 加法器和减法器设计

为了便于理解，我们以无符号数为例讨论二进制数的加法运算。首先考虑一位二进制数加一位二进制数：$0+0=0$，$0+1=1$，$1+0=1$，$1+1=0$ (进位位

表 3.2　4 位二进制数在不同的表示方法下的十进制数值

二进制数	无符号数	补码表示	符号-绝对值	移码 (偏移值 7)
0000	0	0	+0	−7
0001	1	+1	+1	−6
0010	2	+2	+2	−5
0011	3	+3	+3	−4
0100	4	+4	+4	−3
0101	5	+5	+5	−2
0110	6	+6	+6	−1
0111	7	+7	+7	0
1000	8	−8	−0	+1
1001	9	−7	−1	+2
1010	10	−6	−2	+3
1011	11	−5	−3	+4
1100	12	−4	−4	+5
1101	13	−3	−5	+6
1110	14	−2	−6	+7
1111	15	−1	−7	+8

为 1)。只完成两个一位二进制数相加的电路称为半加器 (Half-Adder)。有半加器就有全加器 (Full-Adder)。全加器考虑低位来的进位。

图 3.3 是两个 4 位二进制数相加的情况，进位位用小号字体标出。与十进制数相加类似，我们从最低位开始计算：$1 + 1 = 0$ (进位位为 1)；计算下一位时，我们不仅要对两个一位二进制数相加，而且也要加上从低位来的进位：$1 + 1 + 1 = 1$ (进位位为 1)。我们用信号 a 和 b 表示原来的两个一位二进制数、ci 表示从低位来的进位、s 表示相加的一位结果 (和)、co 表示相加时产生的进位。



图 3.3　4 位二进制数相加

表 3.3 是全加器的真值表，应该不难理解。有了真值表，问题就变得简单了：写出输出信号的逻辑表达式、根据表达式画出逻辑图再做功能仿真。输出信号的逻辑表达式如下。

$$s = \bar{a}\,\bar{b}\,ci + \bar{a}\,b\,\overline{ci} + a\,\bar{b}\,\overline{ci} + a\,b\,ci$$
$$co = a\,b + a\,ci + b\,ci$$

表 3.3 全加器真值表

输入信号			输出信号		注 释
a	b	ci	co	s	
0	0	0	0	0	0 + 0 + 0 = 0 0
0	0	1	0	1	0 + 0 + 1 = 0 1
0	1	0	0	1	0 + 1 + 0 = 0 1
0	1	1	1	0	0 + 1 + 1 = 1 0
1	0	0	0	1	1 + 0 + 0 = 0 1
1	0	1	1	0	1 + 0 + 1 = 1 0
1	1	0	1	0	1 + 1 + 0 = 1 0
1	1	1	1	1	1 + 1 + 1 = 1 1

全加器的电路图见图 3.4。画电路图时，建议大家尽量用名字来标注各逻辑门的输入输出信号线，这样检查起来比较方便。由于 $s = \bar{a}\,\bar{b}\,ci + \bar{a}\,b\,\overline{ci} + a\,\bar{b}\,\overline{ci} + a\,b\,ci = a \oplus b \oplus ci$，我们也可以用异或门产生输出信号 s (图 3.5)。

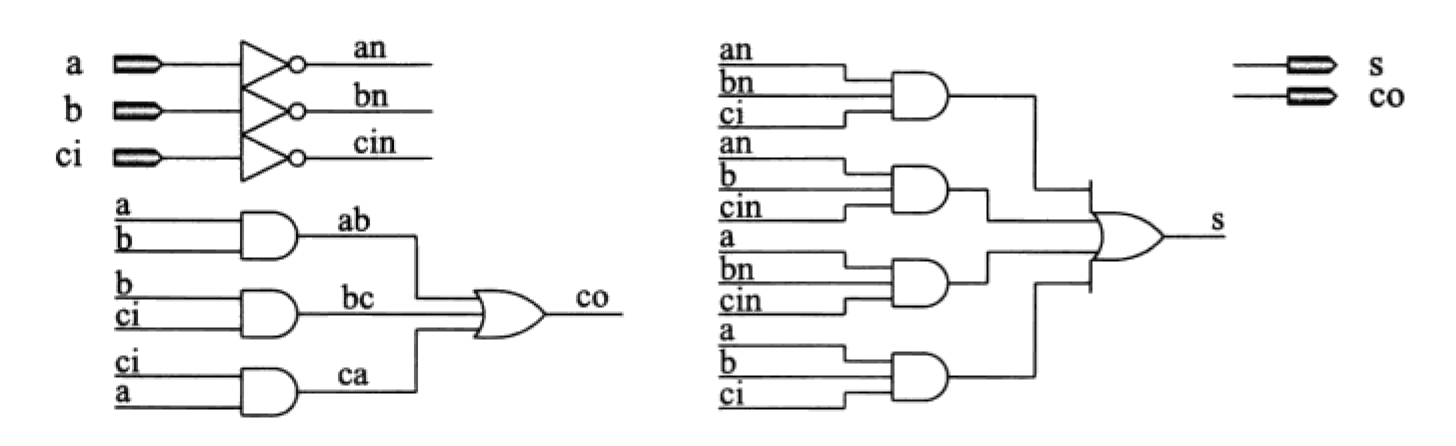


图 3.4 全加器逻辑电路图

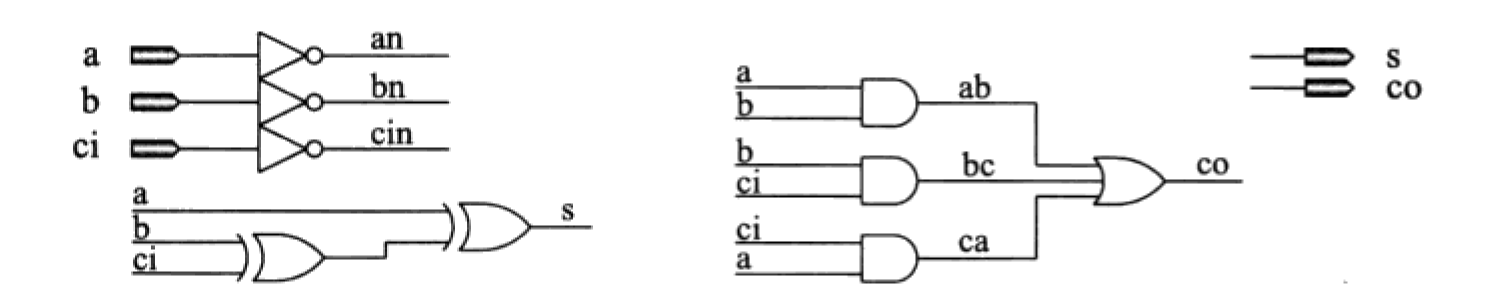


图 3.5 全加器逻辑电路图 (使用异或门)

我们给出全加器三种风格的 Verilog HDL 代码。第一种逻辑门级风格的代码与逻辑图 (图 3.5) 基本相同，只是我们在代码中使用了 3 输入的异或门。

```
module fa_structural (a,b,ci,s,co); // structural style
    input a,b,ci;
    output s,co;
    wire ab, bc, ca;
```

```
    xor (s,a,b,ci);
    and (ab,a,b);
    and (bc,b,ci);
    and (ca,ci,a);
    or  (co,ab,bc,ca);
endmodule
```

第二种逻辑表达式风格的代码如下所示。注意逻辑表达式中的与、或和异或操作在 Verilog HDL 代码中分别用 &、| 和 ^ 表示。该代码也是我们在以后的叙述经常要用到的代码，因此我们使用了比较简单的模块名：add1。

```
module add1 (a,b,ci,s,co); // dataflow style
    input a,b,ci;
    output s,co;
    assign s  = a ^ b ^ ci;
    assign co = (a & b) | (b & ci) | (ci & a);
endmodule
```

如果你不想使用这种逻辑表达式风格的描述，第三种功能描述风格的代码将会满足你的愿望，其中 {co, s} 表示的是两位二进制数。在以后的叙述中，我们基本上使用第二种和第三种风格的代码，不用或尽量少用第一种风格的代码。

```
module fa_behavioral (a,b,ci,s,co); // behavioral style
    input a,b,ci;
    output s,co;
    assign {co,s} = a + b + ci;
endmodule
```

图 3.6 是全加器逻辑电路的仿真结果。由于以上三段代码实现相同的功能，它们的仿真结果当然也相同。

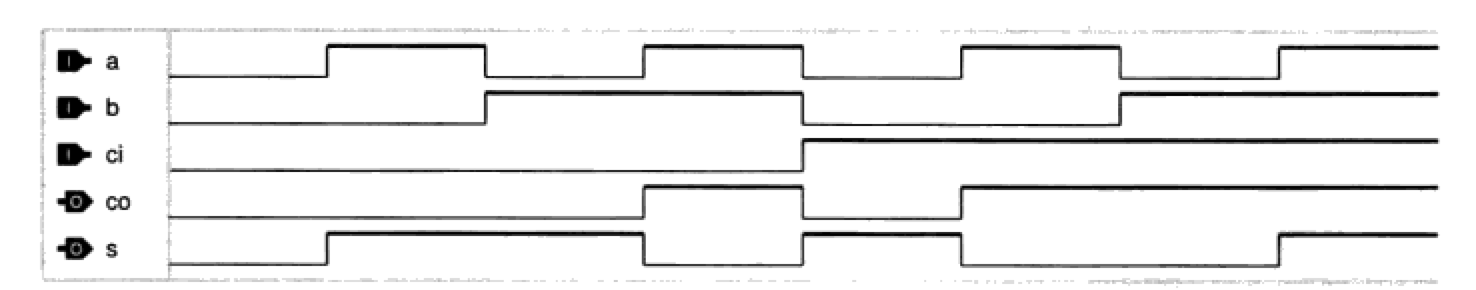


图 3.6 全加器逻辑电路的仿真结果

有了一位全加器，我们可以着手进行多位二进制数加法器的设计工作了。图 3.7 是一个 4 位加法器，它使用了 4 个全加器，低位的进位输出连接到高位的进位输入。

以下是 4 位加法器的 Verilog HDL 代码，实现图 3.7 中的电路。这里我们调用了全加器 add1 模块。一个 3 位的中间变量 cout 对应第 0 到第 2 位的进位输出。

```
module add4 (a,b,ci,s,co);
    input [3:0] a,b;
```

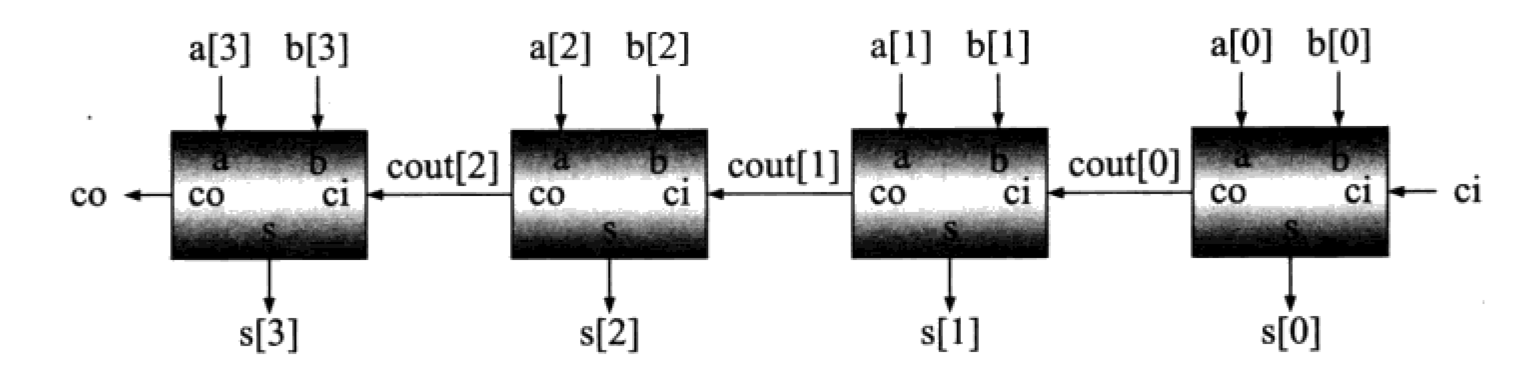


图 3.7 4 位二进制数逐位进位加法器

```
    input ci;
    output [3:0] s;
    output co;
    wire [2:0] cout;
    add1 adder0 (a[0], b[0],      ci, s[0], cout[0]);
    add1 adder1 (a[1], b[1], cout[0], s[1], cout[1]);
    add1 adder2 (a[2], b[2], cout[1], s[2], cout[2]);
    add1 adder3 (a[3], b[3], cout[2], s[3], co);
endmodule
```

现在我们回过头来看图 3.3 给出的加法结果：$0111 + 0011 = 1010$。如果我们认为它们都是无符号数，则该计算相当于十进制的 $7 + 3 = 10$。如果是补码表示的带符号数，这时的结果 1010 是十进制的 -6 (见表 3.2)。相加结果也应该是十进制的 10，但 4 位补码表示的二进制数只能表示到 7。我们称其为结果上溢 (Overflow)，即结果太大，超出了可以表示的数的范围。

我们再看以下的例子。同样是 $0111 + 1011 = 0010$，无符号数相加时结果溢出，而带符号数相加时结果没有溢出。

无符号数相加：		带符号数相加：	
0 1 1 1	(7)	0 1 1 1	(+7)
+ 1 0 1 1	(11)	+ 1 0 1 1	(−5)
0 0 1 0	(2)	0 0 1 0	(+2)

在补码表示当中，由一个数求它的负数的算法如下：首先对该数各位取反，然后再把取反后的数加 1。即：$-b = \overline{b} + 1$。例如：

0 1 1 1	(+7)	1 0 0 1	(−7)
1 0 0 0	(取反)	0 1 1 0	(取反)
+ 0 0 0 1	(加 1)	+ 0 0 0 1	(加 1)
1 0 0 1	(−7)	0 1 1 1	(+7)

在上例中，我们称 1001 是 0111 的补码；同样，0111 是 1001 的补码，即互为补码。此处的补码的全称是“2 的补码”。为什么这样称呼，理由如下。假定在最高位的右面有一个小数点存在，则一个数的补码可以从 2 减去这个数得到，或者说，一个数加上它的补码等于 2：

$$\begin{array}{r} 10.000 \\ -\ 0.111 \\ \hline 1.001 \end{array} \begin{array}{l} (2) \\ \\ (0.111\text{的补码}) \end{array} \qquad \begin{array}{r} 0.111 \\ +1.001 \\ \hline 10.000 \end{array} \begin{array}{l} \\ (0.111\text{的补码}) \\ (2) \end{array}$$

为什么取反加 1 就会有这样的效果呢？这是因为一个数加上它的"反"等于全 1，再加 1 变成全 0 并在最高位产生进位，也就是考虑小数点时的 2。

使用 $-b = \overline{b} + 1$，我们可以用设计好的加法器来实现减法操作：

$$a - b = a + (-b) = a + \overline{b} + 1$$

图 3.8 是 4 位二进制数加减法器的电路图。图中的输入信号 sub 为 1 时做减法，为 0 时做加法。做减法时异或门刚好用来对 b 取反，这是因为：$b \oplus 1 = \overline{b}$，而 $b \oplus 0 = b$。

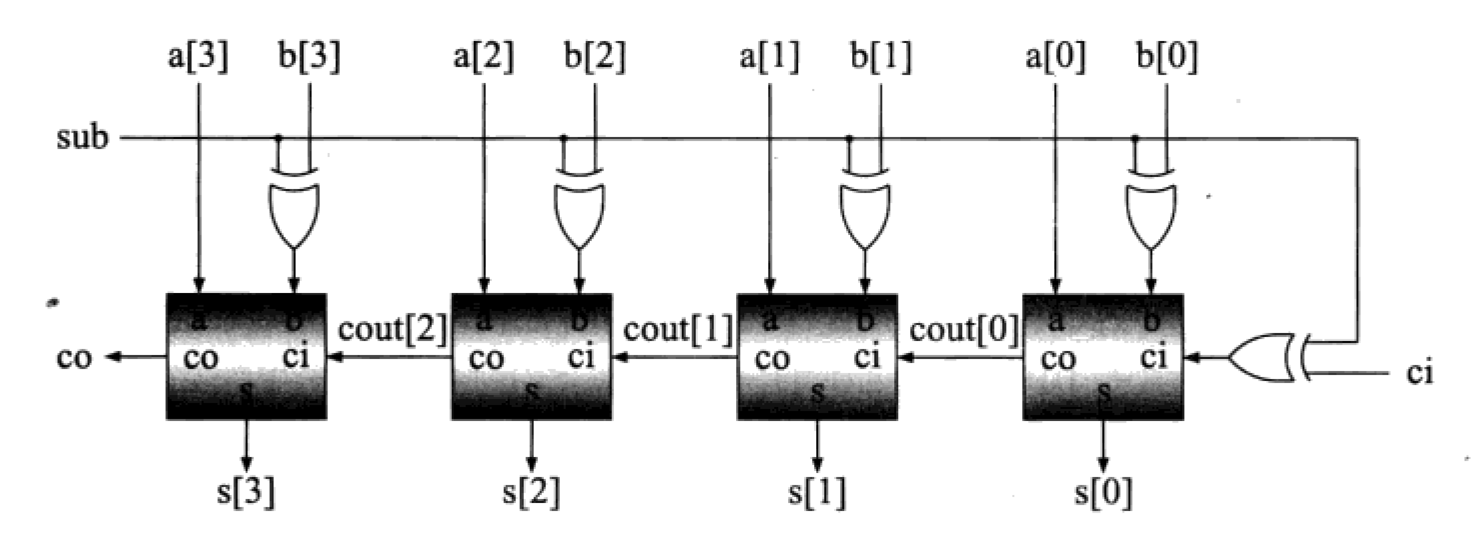


图 3.8 4 位二进制数加减法器

图 3.8 电路的 Verilog HDL 代码如下。对 ci 及 b 和 sub 的异或输出分别为 cib 及 bb，式中 {4{sub}} 是把 sub 扩展成 4 位。

```
module addsub4 (a,b,ci,sub,s,co);
    input [3:0] a,b;
    input ci;
    input sub; // 1: sub; 0: add
    output [3:0] s;
    output co;

    wire cib = ci ^ sub;
    wire [3:0] bb = b ^ {4{sub}};
    wire [2:0] cout;
    add1 adder0  ( a[0], bb[0],     cib, s[0], cout[0]);
    add1 adder1  ( a[1], bb[1], cout[0], s[1], cout[1]);
    add1 adder2  ( a[2], bb[2], cout[1], s[2], cout[2]);
    add1 adder3  ( a[3], bb[3], cout[2], s[3], co);
endmodule
```

以上代码的仿真结果见图 3.9。再说一遍，不管是无符号数还是补码表示的带符号数，做加减运算时并无区别，区别在于对结果如何解释及如何使用。判断结果是否溢出的问题，见本章最后的练习题。

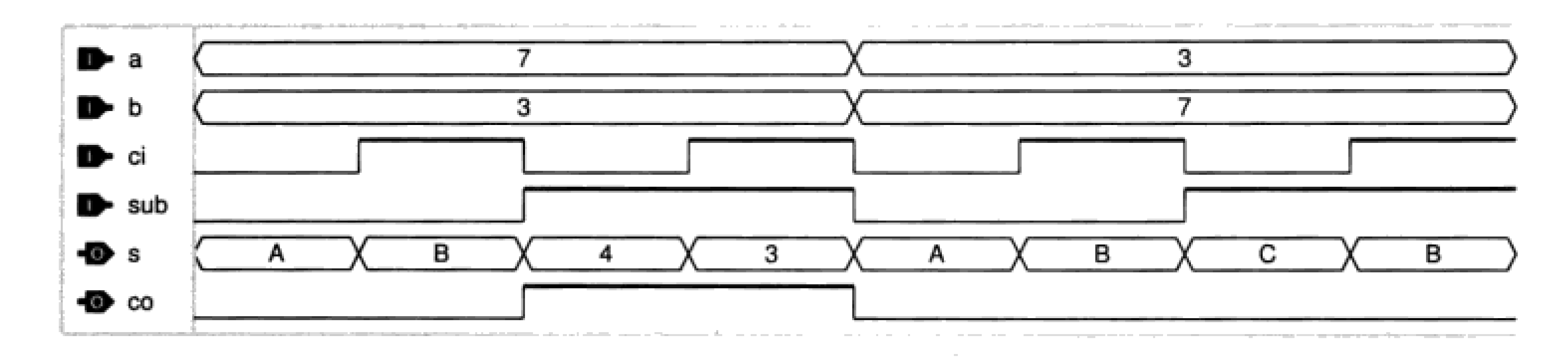


图 3.9 加减法器电路的仿真结果

3.2.2 先行进位加法器设计

以上讨论的逐位进位加法器用英文称为 Ripple Adder。如果我们有一个 32 位的这样的加法器，进位将从输入的 ci 逐位地传递到最高位的进位输出 co，途中穿越 32 个全加器。

我们知道，电路是有延迟的，这样的长途旅行是要花时间的。为了加快加法器的运算速度，我们必须想办法能尽快地产生每位的进位位。快速产生进位位的方法有很多种，以下仅介绍一种简单的树型进位产生算法。由于

$$\begin{aligned} c[i+1] &= a[i]\,b[i] + a[i]\,c[i] + b[i]\,c[i] \\ &= a[i]\,b[i] + (a[i] + b[i])\,c[i] \\ &= g[i] + p[i]\,c[i] \end{aligned}$$

其中，$g[i] = a[i]\,b[i]$ 称为进位产生函数，$p[i] = a[i] + b[i]$ 称为进位传递函数，则

$$\begin{aligned} c[i+1] = g[i] &+ p[i]\,g[i-1] \\ &+ p[i]\,p[i-1]\,g[i-2] \\ &+ \cdots \\ &+ p[i]\,p[i-1]\cdots p[1]\,g[0] \\ &+ p[i]\,p[i-1]\cdots p[1]\,p[0]\,c[0] \end{aligned}$$

从理论上讲，由 $c[i+1]$ 的表达式我们可以并行产生所有位的进位输入，但是这样的电路太复杂，VLSI 实现起来不太现实。因此我们必须找出简单的且有规则的电路来产生进位输入。

我们采用树型结构分层次产生进位位。表 3.4 列出了 8 位先行进位加法器的进位产生的情况。图 3.10 是 4 位先行进位加法器的电路图。

以下我们给出 32 位先行进位加法器的 Verilog HDL 代码。首先是一位加法器的代码，它是图 3.10 中最上层的模块的代码。除了产生一位加法结果 s 外，它也生成进位产生函数 g 和进位传递函数 p。

表 3.4 8 位先行进位加法器

进 位	产生函数和传递函数
第一组 $c[0] = c_in$ $c[1] = g[0] + p[0]\,c[0]$	$G[1,0] = g[1] + p[1]\,g[0]$ $P[1,0] = p[1]\,p[0]$
第二组 $c[2] = G[1,0] + P[1,0]\,c[0]$ $c[3] = g[2] + p[2]\,c[2]$	$G[3,2] = g[3] + p[3]\,g[2]$ $P[3,2] = p[3]\,p[2]$
第一、二组 ⇒	$G[3,0] = G[3,2] + P[3,2]\,G[1,0]$ $P[3,0] = P[3,2]\,P[1,0]$
第三组 $c[4] = G[3,0] + P[3,0]\,c[0]$ $c[5] = g[5] + p[5]\,c[4]$	$G[5,4] = g[5] + p[5]\,g[4]$ $P[5,4] = p[5]\,p[4]$
第四组 $c[6] = G[5,4] + P[5,4]\,c[4]$ $c[7] = g[7] + p[7]\,c[6]$	$G[7,6] = g[7] + p[7]\,g[6]$ $P[7,6] = p[7]\,p[6]$
第三、四组 ⇒	$G[7,4] = G[7,6] + P[7,6]\,G[5,4]$ $P[7,4] = P[7,6]\,P[5,4]$
第一、二、三、四组 ⇒	$G[7,0] = G[7,4] + P[7,4]\,G[3,0]$ $P[7,0] = P[7,4]\,P[3,0]$
$c[8] = G[7,0] + P[7,0]\,c[0]$	

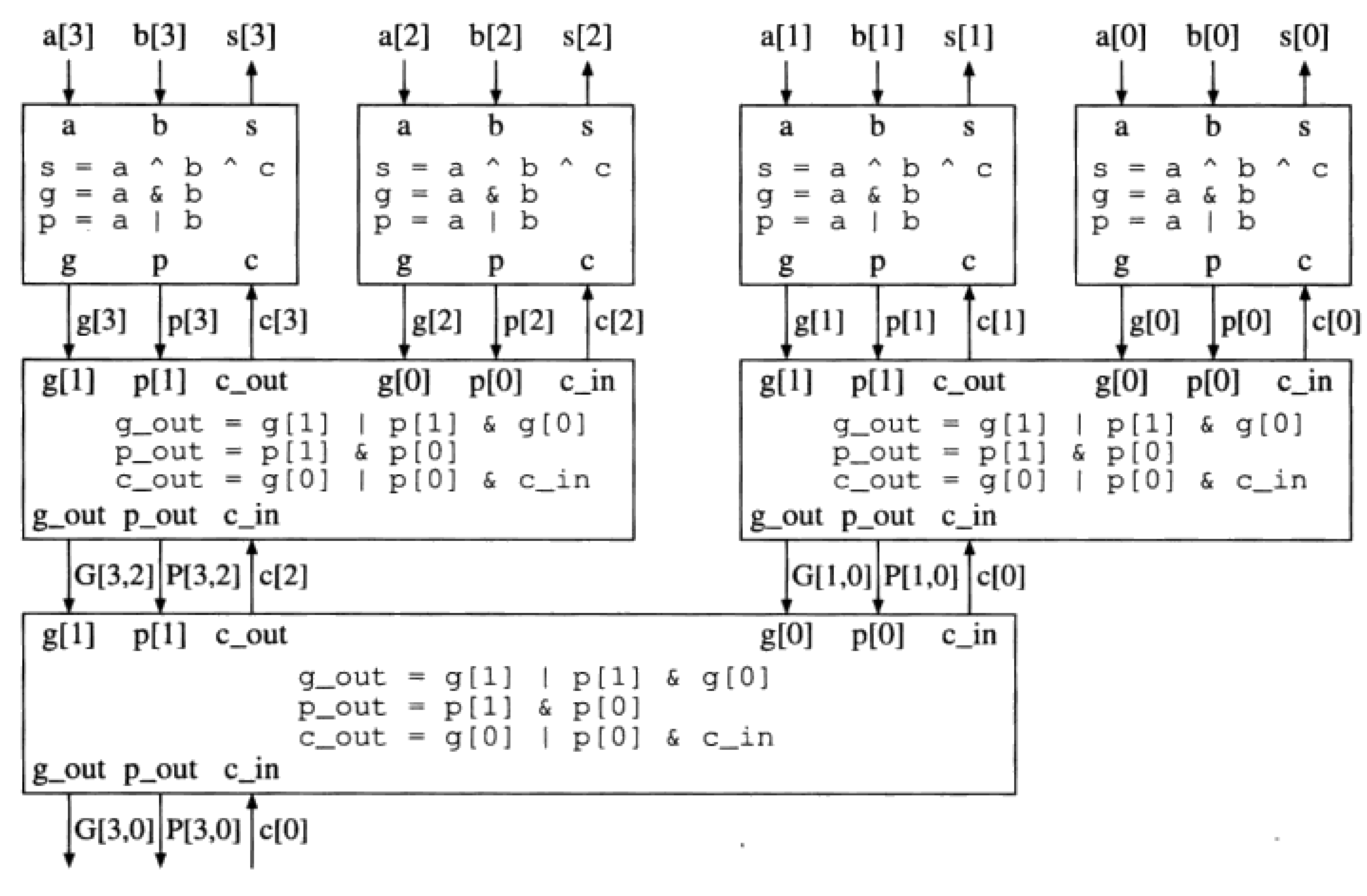


图 3.10 4 位先行进位加法器

```
module add (a,b,c,g,p,s);
    input a,b,c;
    output g,p,s;
    assign s = a ^ b ^ c;
    assign g = a & b;
```

```
    assign p = a | b;
endmodule
```

图 3.10 中间一层的两个模块和最下一层的模块是相同的电路，它的代码如下。除了产生一位进位位 c[i] 外，它也生成进位产生函数 G 和进位传递函数 P。我们这里称它为 GP 生成器。

```
module g_p (g,p,c_in, g_out,p_out,c_out);
    input [1:0] g,p;
    input c_in;
    output g_out,p_out,c_out;
    assign g_out = g[1] | p[1] & g[0];
    assign p_out = p[1] & p[0];
    assign c_out = g[0] | p[0] & c_in;
endmodule
```

有了以上两个模块，我们可以设计一个两位的先行进位加法器了。它使用了两个一位加法器和一个 GP 生成器。这三个模块之间的连接关系请参照图 3.10。

```
module cla_2 (a,b,c_in, g_out,p_out,s);
    input [1:0] a,b;
    input c_in;
    output g_out,p_out;
    output [1:0] s;
    wire [1:0] g,p;
    wire c_out;
    add add0 (a[0],b[0],c_in,  g[0],p[0],s[0]);
    add add1 (a[1],b[1],c_out, g[1],p[1],s[1]);
    g_p g_p0 (g,p,c_in, g_out,p_out,c_out);
endmodule
```

使用两个两位先行进位加法器和一个 GP 生成器，我们可以设计一个 4 位先行进位加法器。

```
module cla_4 (a,b,c_in,g_out,p_out,s);
    input [3:0] a,b;
    input c_in;
    output g_out,p_out;
    output [3:0] s;
    wire [1:0] g,p;
    wire c_out;
    cla_2 cla0 (a[1:0],b[1:0],c_in,  g[0],p[0],s[1:0]);
    cla_2 cla1 (a[3:2],b[3:2],c_out, g[1],p[1],s[3:2]);
    g_p  g_p0 (g,p,c_in, g_out,p_out,c_out);
endmodule
```

使用两个 4 位先行进位加法器和一个 GP 生成器，我们可以设计一个 8 位先行进位加法器。

```
module cla_8 (a,b,c_in,g_out,p_out,s);
    input [7:0] a,b;
    input c_in;
    output g_out,p_out;
    output [7:0] s;
    wire [1:0] g,p;
    wire c_out;
    cla_4 cla0 (a[3:0],b[3:0],c_in,  g[0],p[0],s[3:0]);
    cla_4 cla1 (a[7:4],b[7:4],c_out, g[1],p[1],s[7:4]);
    g_p  g_p0 (g,p,c_in, g_out,p_out,c_out);
endmodule
```

使用两个 8 位先行进位加法器和一个 GP 生成器，我们可以设计一个 16 位先行进位加法器。

```
module cla_16 (a,b,c_in,g_out,p_out,s);
    input [15:0] a,b;
    input c_in;
    output g_out,p_out;
    output [15:0] s;
    wire [1:0] g,p;
    wire c_out;
    cla_8 cla0 (a[7:0], b[7:0], c_in,  g[0],p[0],s[7:0]);
    cla_8 cla1 (a[15:8],b[15:8],c_out, g[1],p[1],s[15:8]);
    g_p  g_p0 (g,p,c_in, g_out,p_out,c_out);
endmodule
```

使用两个 16 位先行进位加法器和一个 GP 生成器，我们可以设计一个 32 位先行进位加法器。

```
module cla_32 (a,b,c_in,g_out,p_out,s);
    input [31:0] a,b;
    input c_in;
    output g_out,p_out;
    output [31:0] s;
    wire [1:0] g,p;
    wire c_out;
    cla_16 cla0 (a[15:0], b[15:0], c_in,  g[0],p[0],s[15:0]);
    cla_16 cla1 (a[31:16],b[31:16],c_out, g[1],p[1],s[31:16]);
    g_p  g_p0 (g,p,c_in, g_out,p_out,c_out);
endmodule
```

以下是 32 位先行进位加法器的最后的代码，除了产生 32 位加法结果，它也产生最后的进位输出。

```
module cla32 (a,b,ci,s,co);
    input [31:0] a,b;
```

```
    input ci;
    output [31:0] s;
    output co;
    wire g_out,p_out;
    cla_32 cla (a,b,ci,g_out,p_out,s);
    assign co = g_out | p_out & ci;
endmodule
```

图 3.11 是 32 位先行进位加法器 cla32 的仿真结果。

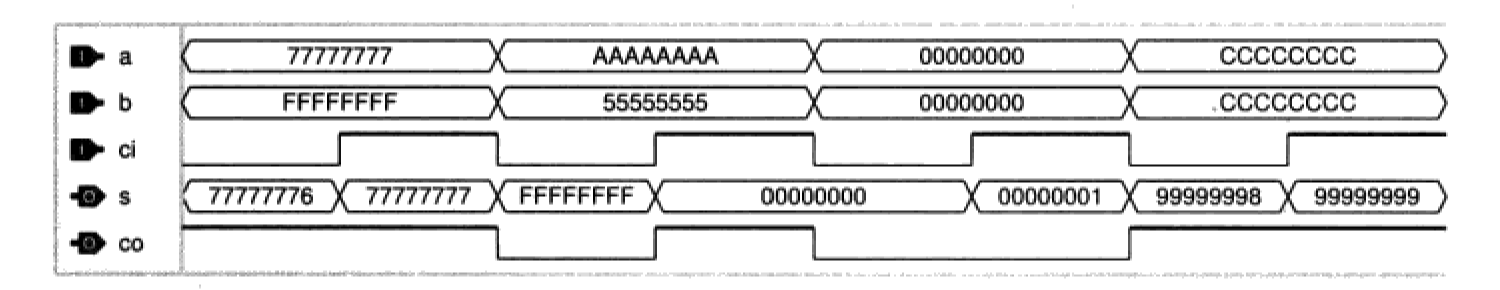


图 3.11 32 位先行进位加法器电路的仿真结果

3.3 乘法算法及 Verilog HDL 实现

本节介绍无符号数和带符号数的乘法算法。首先介绍基本的乘法算法，然后介绍 Wallace 树型乘法算法并给出 Verilog HDL 的实现。

3.3.1 无符号数乘法器设计

与十进制乘法计算一样，二进制的乘法可以用加法和移位操作来完成，例如：

$$
\begin{array}{rcccccccl}
 & & & & 1 & 1 & 1 & 0 & (14_{10}) \\
 & & & \times & 1 & 0 & 1 & 0 & (10_{10}) \\
\hline
 & & & & 0 & 0 & 0 & 0 & \\
 & & & 1 & 1 & 1 & 0 & & \\
 & & 0 & 0 & 0 & 0 & & & \\
+ & 1 & 1 & 1 & 0 & & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & (140_{10})
\end{array}
$$

两位二进制数相乘和逻辑与操作相同。我们可以用全加器阵列实现上述加法，也可以用循环迭代的方法实现。以下是两个 16 位无符号二进制数用迭代方法实现乘法操作的 C 语言版本，也是通过加法和移位操作来完成的。

```
#include <stdio.h>
unsigned int mul16 (unsigned int x, unsigned int y) {
    unsigned int a, b, c;
    unsigned int i; // counter
```

```
    a = x; // multiplicand
    b = y; // multiplier
    c = 0; // product
    for (i = 0; i < 16; i++) { // for 16 bits
        if ((b & 1) == 1) { // LSB of b is 1
            c += a; // c = c + a
        }
        a = a << 1; // shift a 1-bit left
        b = b >> 1; // shift b 1-bit right
    }
    return(c);       // return product
}

main() {
    unsigned int x,y;
    fprintf(stderr,"input 1st 16-bit unsigned integer in hex: ");
    fscanf(stdin,"%x",&x);
    fprintf(stderr,"input 2nd 16-bit unsigned integer in hex: ");
    fscanf(stdin,"%x",&y);
    x &= 0xffff;
    y &= 0xffff;
    fprintf(stderr,"%04x * %04x = %08x\n", x, y, mul16(x, y));
}
```

编译及运行的例子如下所示：

```
[yamin@localhost cpu]$ gcc mul.c -o mul
[yamin@localhost cpu]$ ./mul
input 1st 16-bit unsigned integer in hex: c9ae
input 2nd 16-bit unsigned integer in hex: f6e5
c9ae * f6e5 = c2819ca6
[yamin@localhost cpu]$
```

作为练习题，请读者用 Verilog HDL 实现上述 mul16 所完成的操作。

3.3.2 带符号数乘法器设计

设有两个补码表示的 8 位带符号数 A_8 和 B_8，试计算 $Z_{16} = A_8 \times B_8$。

$$A_8 = a_7a_6a_5a_4a_3a_2a_1a_0 = -a_7 \times 2^7 + \sum_{i=0}^{6} a_i \times 2^i = -a_7 \times 2^7 + A_7$$

$$B_8 = b_7b_6b_5b_4b_3b_2b_1b_0 = -b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i = -b_7 \times 2^7 + B_7$$

其中，A_7 和 B_7 是无符号数。由于 $(a + b)(x + y) = ax + ay + bx + by$，我们有 $Z_{16} = A_8 \times B_8 = (-a_7 \times 2^7 + A_7) \times (-b_7 \times 2^7 + B_7) = a_7 \times b_7 \times 2^{14} + (-a_7 \times$

$B_7) \times 2^7 + (-b_7 \times A_7) \times 2^7 + A_7 \times B_7$。其中第 1 项 $a_7 \times b_7$ 和第 4 项 $A_7 \times B_7$ 与无符号数乘相同，而第 2 项和第 3 项为负，如图 3.12 所示[32]。

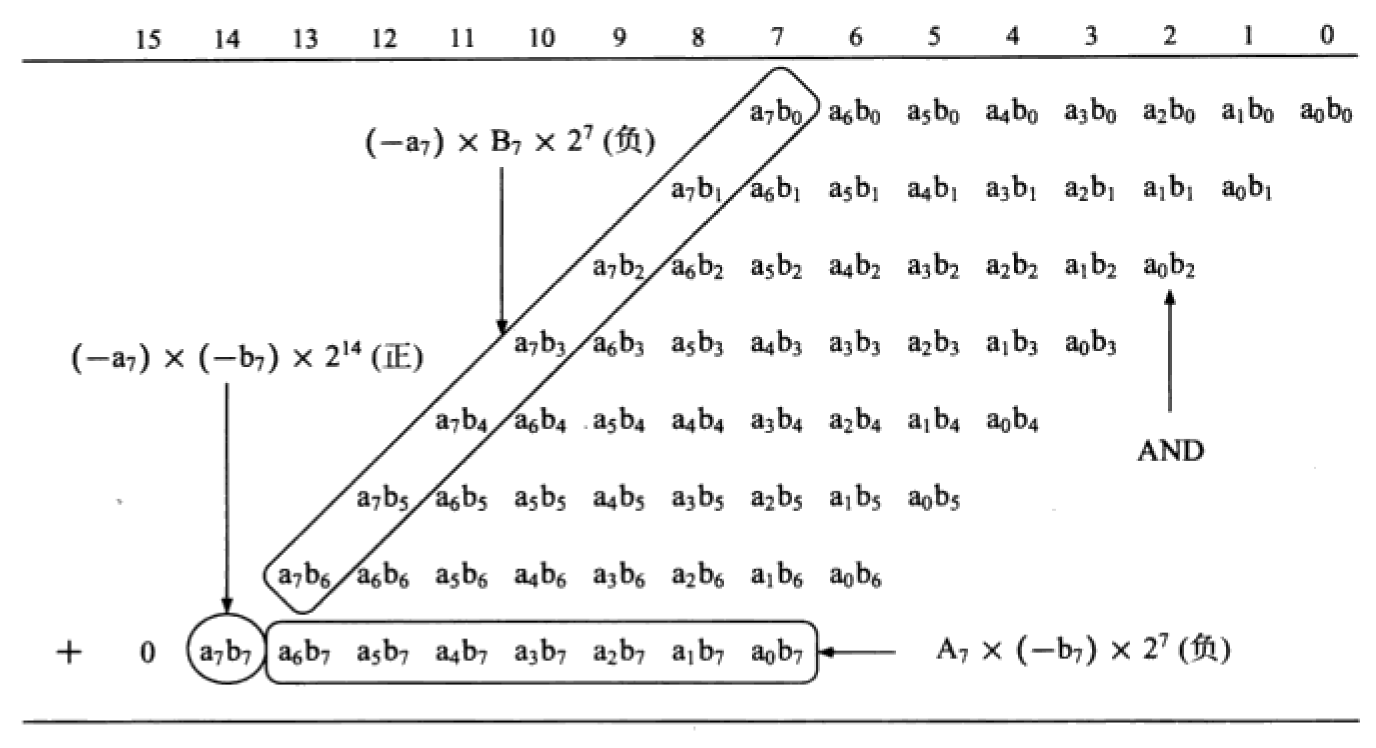


图 3.12 带符号数乘法：a_7和 b_7 为负

写成二进制位的形式：

$$a_7 \times B_7 = 0\ \ 0\ \ a_7b_6\ \ a_7b_5\ \ a_7b_4\ \ a_7b_3\ \ a_7b_2\ \ a_7b_1\ \ a_7b_0$$
$$b_7 \times A_7 = 0\ \ 0\ \ a_6b_7\ \ a_5b_7\ \ a_4b_7\ \ a_3b_7\ \ a_2b_7\ \ a_1b_7\ \ a_0b_7$$

我们已经知道 $-x = \bar{x} + 1$。对这两项同时取反加 1 后再相加，我们有

	15	14	13	12	11	10	9	8	7
	1	1	$\overline{a_7b_6}$	$\overline{a_7b_5}$	$\overline{a_7b_4}$	$\overline{a_7b_3}$	$\overline{a_7b_2}$	$\overline{a_7b_1}$	$\overline{a_7b_0}$
	1	1	$\overline{a_6b_7}$	$\overline{a_5b_7}$	$\overline{a_4b_7}$	$\overline{a_3b_7}$	$\overline{a_2b_7}$	$\overline{a_1b_7}$	$\overline{a_0b_7}$
	0	0	0	0	0	0	0	0	1
+	0	0	0	0	0	0	0	0	1

化简后变为：

	15	14	13	12	11	10	9	8	7
	0	0	$\overline{a_7b_6}$	$\overline{a_7b_5}$	$\overline{a_7b_4}$	$\overline{a_7b_3}$	$\overline{a_7b_2}$	$\overline{a_7b_1}$	$\overline{a_7b_0}$
	0	0	$\overline{a_6b_7}$	$\overline{a_5b_7}$	$\overline{a_4b_7}$	$\overline{a_3b_7}$	$\overline{a_2b_7}$	$\overline{a_1b_7}$	$\overline{a_0b_7}$
+	1	0	0	0	0	0	0	1	0

我们把两个 1 放在合适的位置，如图 3.13 所示，再把所有乘积项相加，就可得到带符号数的相乘结果。

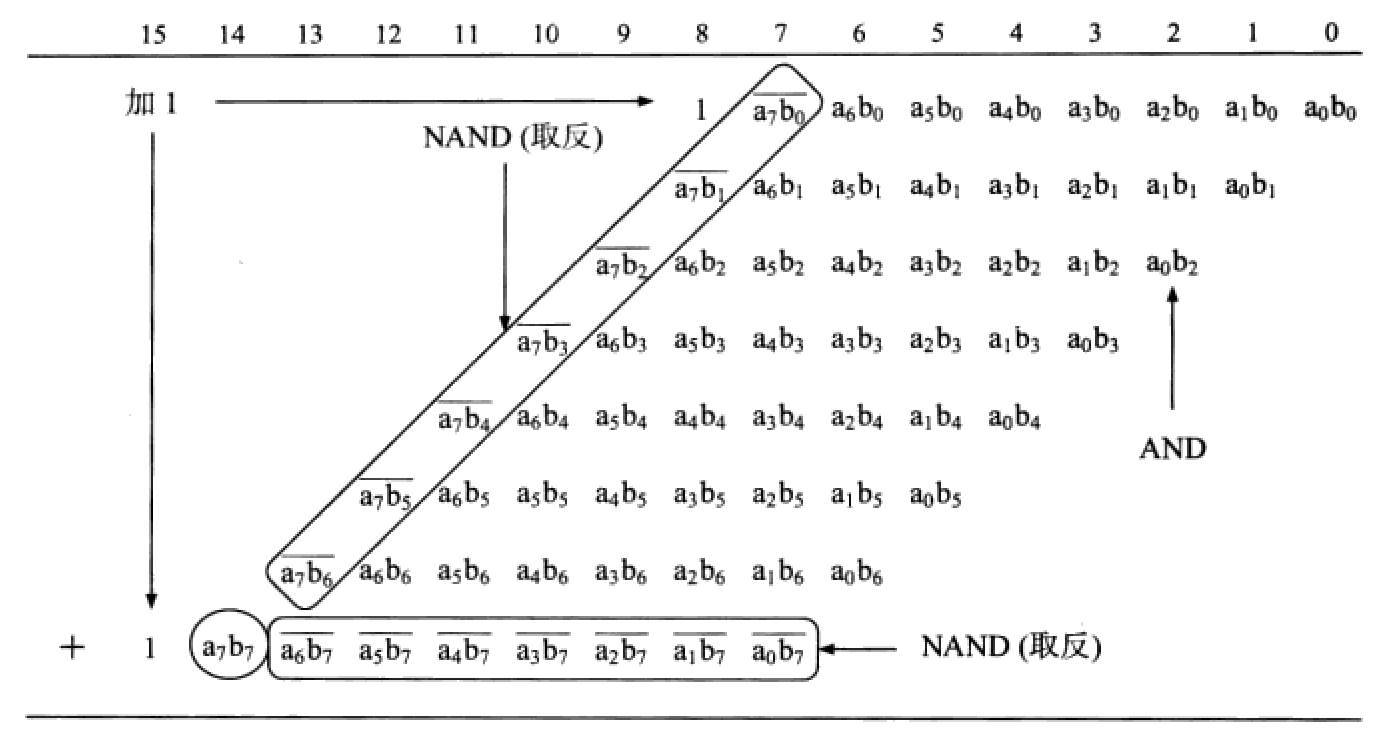


图 3.13 带符号数乘法：取反加 1

带符号数乘法器的非常直观的 Verilog HDL 代码如下。前面部分乘积项的产生"相当于"全部用与门，后面部分相加之前再取反。我们在相加语句中加了括号，以实现并行相加。如果不加括号，有可能生成的电路是顺序相加。

```
module mul_signed (a,b,z);
    input  [7:0] a,b;
    output [15:0] z;
    wire [7:0] ab0 = b[0]? a : 8'b0 ;
    wire [7:0] ab1 = b[1]? a : 8'b0 ;
    wire [7:0] ab2 = b[2]? a : 8'b0 ;
    wire [7:0] ab3 = b[3]? a : 8'b0 ;
    wire [7:0] ab4 = b[4]? a : 8'b0 ;
    wire [7:0] ab5 = b[5]? a : 8'b0 ;
    wire [7:0] ab6 = b[6]? a : 8'b0 ;
    wire [7:0] ab7 = b[7]? a : 8'b0 ;
    assign z = (({8'b1,~ab0[7],ab0[6:0]}          +
                  {7'b0,~ab1[7],ab1[6:0],1'b0})   +
                 ({6'b0,~ab2[7],ab2[6:0],2'b0}    +
                  {5'b0,~ab3[7],ab3[6:0],3'b0})) +
                (({4'b0,~ab4[7],ab4[6:0],4'b0}    +
                  {3'b0,~ab5[7],ab5[6:0],5'b0})   +
                 ({2'b0,~ab6[7],ab6[6:0],6'b0}    +
                  {1'b1,ab7[7],~ab7[6:0],7'b0}));
endmodule
```

以下是带符号数乘法器的第二个 Verilog HDL 版本。它直接在前面部分用与非门，经逻辑综合后的电路比第一个版本简单。第一个版本前面部分乘积项的产生经逻辑综合后使用了多路选择器。

```
module mul_signed_v2 (a,b,z);
    input  [7:0] a,b;
    output [15:0] z;
    reg [7:0] a_bi[7:0];
    always @ * begin
        integer i,j;
        for (i = 0; i < 7; i = i + 1)
            for (j = 0; j < 7; j = j + 1)
                a_bi[i][j] = a[i] & b[j];
        for (i = 0; i < 7; i = i + 1)
            a_bi[i][7] = ~(a[i] & b[7]);
        for (j = 0; j < 7; j = j + 1)
            a_bi[7][j] = ~(a[7] & b[j]);
        a_bi[7][7] = a[7] & b[7];
    end
    assign z = ((({8'b1,a_bi[0][7],a_bi[0][6:0]}          +
                 {7'b0,a_bi[1][7],a_bi[1][6:0],1'b0})  +
                ({6'b0,a_bi[2][7],a_bi[2][6:0],2'b0}   +
                 {5'b0,a_bi[3][7],a_bi[3][6:0],3'b0})) +
               ((({4'b0,a_bi[4][7],a_bi[4][6:0],4'b0}  +
                 {3'b0,a_bi[5][7],a_bi[5][6:0],5'b0})  +
                ({2'b0,a_bi[6][7],a_bi[6][6:0],6'b0}   +
                 {1'b1,a_bi[7][7],a_bi[7][6:0],7'b0}));
endmodule
```

两个版本有相同的仿真结果，见图 3.14。最左数据示出的是 $(-1)\times(-1)=1$。

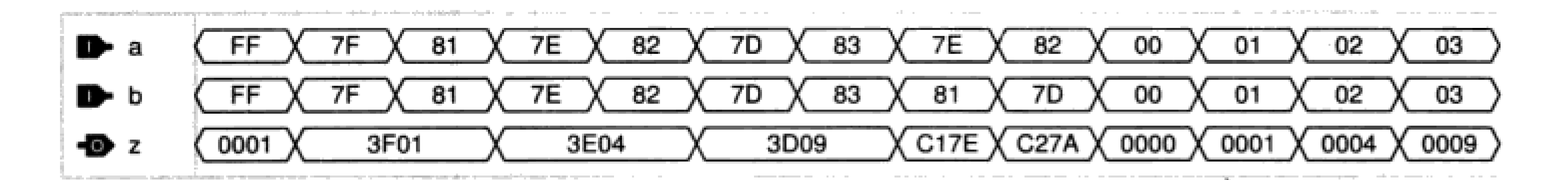


图 3.14 带符号数乘法器的仿真结果

3.3.3 无符号数 Wallace 树型乘法器设计

我们以 8 位乘以 8 位无符号数为例来说明 Wallace Tree 乘法算法，见图 3.15。图中 8 位二进制数 a 和 8 位二进制数 b 相乘产生 64 个乘积项，$a[i]\times b[j]$, $i,j=0,1,\cdots,7$，图中用 * 表示。它们所在的位置构成平行四边形。处在同一列的所有乘积项与从右边过来的进位相加，得到乘积的一位结果。由于 64 个乘积项可以用 64 个逻辑与门同时得到，我们可以用多个全加器对处在同一列的乘积项同时相加。

一个全加器有 3 位输入和两位输出：一位输出是加法结果，另一位是进位。图中的一个长方形代表一个全加器 (内部有两个 * 的长方形可以用半加器实现)。圆圈中的单个乘积项暂时不做任何处理，送到下一级参加运算。

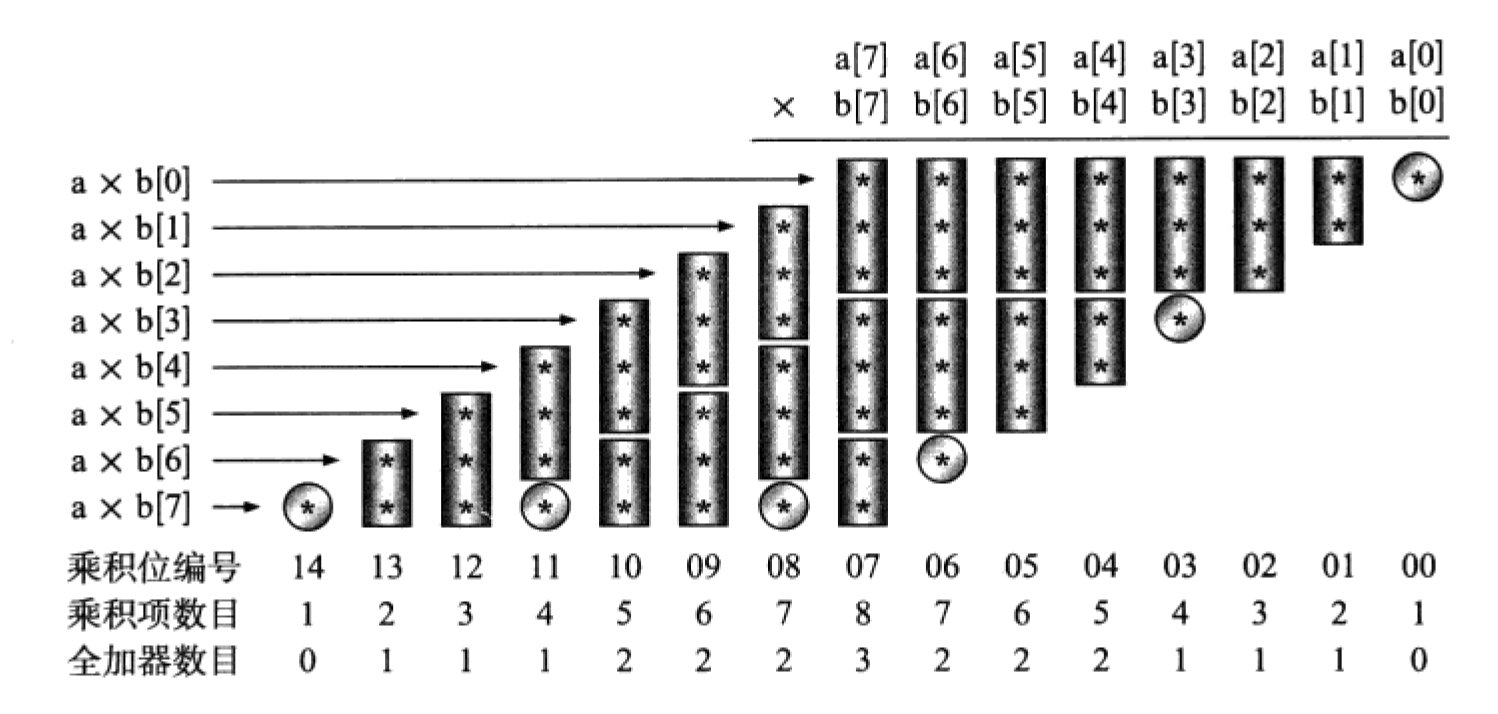


图 3.15　8 × 8 Wallace 树型乘法器乘积项

图 3.16 示出了第 07 位的运算情况。第 07 位有 8 个乘积项，它们分别是 a[7] × b[0]，a[6] × b[1]，a[5] × b[2]，a[4] × b[3]，a[3] × b[4]，a[2] × b[5]，a[1] × b[6] 和 a[0] × b[7]。注意每一项 a 和 b 括号中的数字相加等于 7。

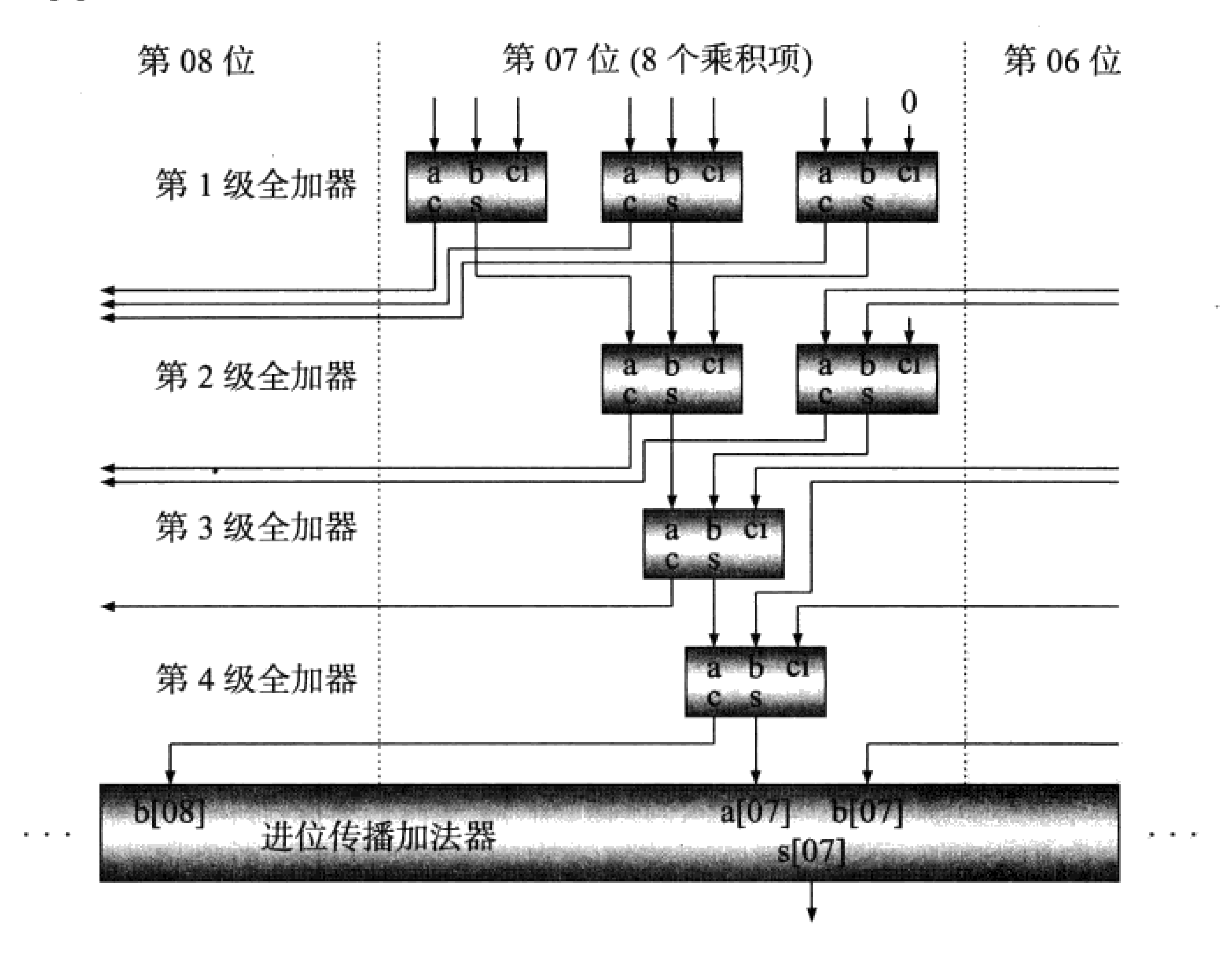


图 3.16　8 × 8 Wallace 树型乘法器第 07 位

在第 1 级，我们使用 3 个全加器，产生 3 位相加结果及 3 位进位。3 位进位送到左边的第 08 位。第 2 级有 5 位要相加，其中的 3 位来自第 1 级，两位是来自右边第

06 位的进位。第 2 级使用两个全加器。第 3 级和第 4 级都只用一个全加器。每一级的延迟等于一个全加器的延迟。我们把所有第 4 级的输出分成 c (Carry) 和 s (Sum) 两组，再把它们用进位传播加法器相加，得到最后的结果 (乘积)。

图 3.17 是 8 × 8 Wallace 树型乘法器总体电路图。图中第 1 行数字是乘积位的编号，第 2 行的数字是相应位的乘积项数量。加法器中的数字是该加法器名称的右半部分，左半部分在图中的最左侧给出。例如，左上角的加法器的名称为 fa1_12_0。

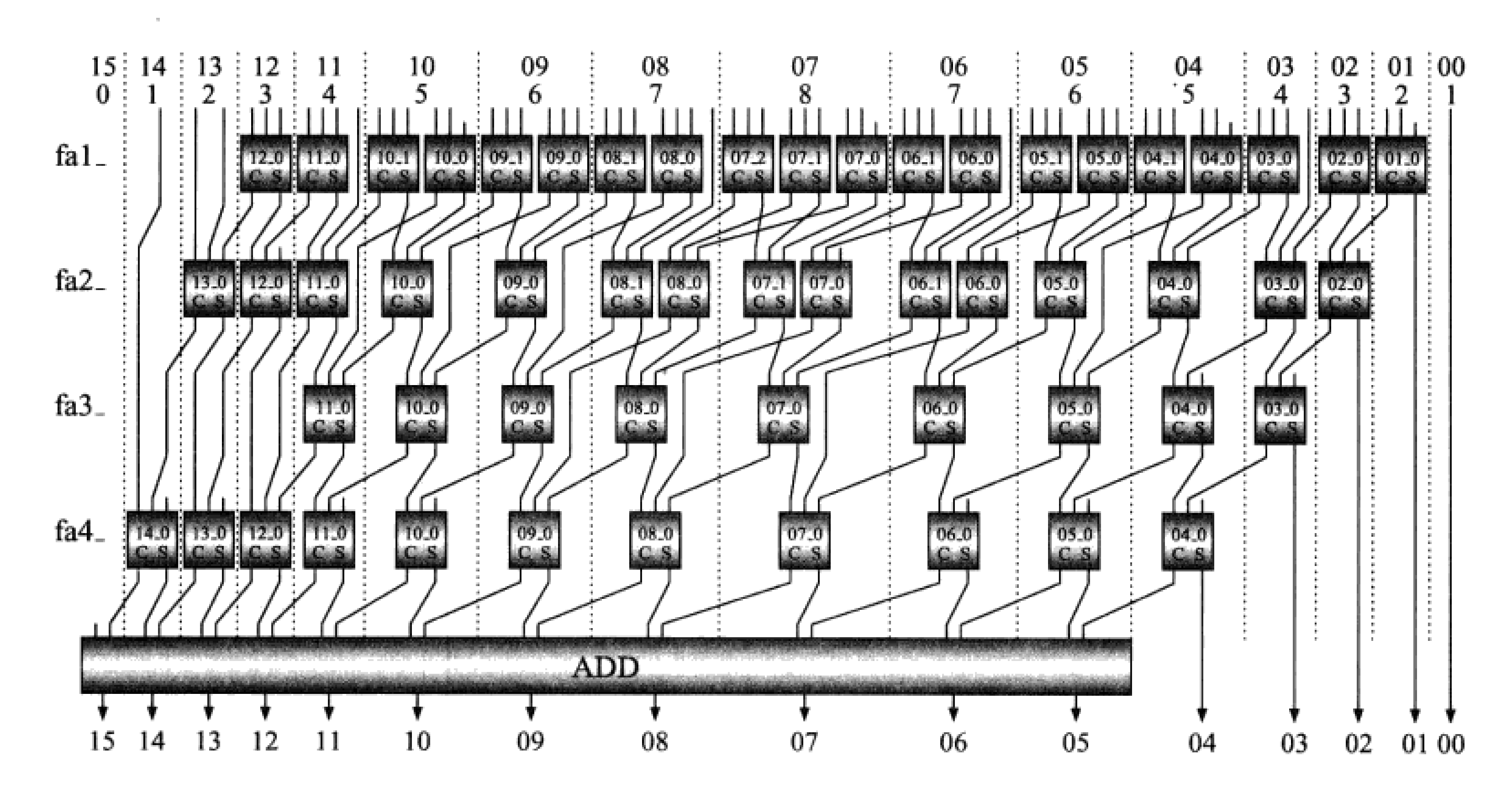


图 3.17　8 × 8 Wallace 树型乘法器

现在我们给出实现 8 × 8 Wallace 树型乘法器的 Verilog HDL 代码。输入是两个 8 位的变量 a 和 b，输出是 16 位 z。首先我们用 64 个与门实现 64 个乘积项 $p[i][j] = a[i] \& b[j]$, $i, j = 0, 1, \cdots, 7$。

```
module wallace_tree8 (a,b,z);
    input  [7:0] a,b;
    output [15:0] z;
    reg [7:0] p[7:0];
    always @ * begin
        integer i,j;
        for (i = 0; i < 8; i = i + 1)
            for (j = 0; j < 8; j = j + 1)
                p[i][j] = a[i] & b[j];
        end
    assign z[0] = p[0][0];
```

以下是第 1 级加法器的代码。add1 (a, b, ci, s, co) 是全加器模块，其中 a，b，ci 是 3 位输入，s 是全加器的和，co 是进位。双斜杠右边的是本级未处理的乘积项，不要忘了把它们送到下一级。

```
parameter zero = 1'b0;
wire [2:0] s1[12:01];
wire [2:0] c1[13:02];
// index 14:   p[7][7]
// index 13:   p[7][6],p[6][7]
add1 fa1_12_0 (p[7][5],p[6][6],p[5][7],s1[12][0],c1[13][0]);
add1 fa1_11_0 (p[7][4],p[6][5],p[5][6],s1[11][0],c1[12][0]);
// index 11:   p[4][7]
add1 fa1_10_1 (p[7][3],p[6][4],p[5][5],s1[10][1],c1[11][1]);
add1 fa1_10_0 (p[4][6],p[3][7],zero,   s1[10][0],c1[11][0]);
add1 fa1_09_1 (p[7][2],p[6][3],p[5][4],s1[09][1],c1[10][1]);
add1 fa1_09_0 (p[4][5],p[3][6],p[2][7],s1[09][0],c1[10][0]);
add1 fa1_08_1 (p[7][1],p[6][2],p[5][3],s1[08][1],c1[09][1]);
// index 08:   p[1][7]
add1 fa1_08_0 (p[4][4],p[3][5],p[2][6],s1[08][0],c1[09][0]);
add1 fa1_07_2 (p[7][0],p[6][1],p[5][2],s1[07][2],c1[08][2]);
add1 fa1_07_1 (p[4][3],p[3][4],p[2][5],s1[07][1],c1[08][1]);
add1 fa1_07_0 (p[1][6],p[0][7],zero,   s1[07][0],c1[08][0]);
add1 fa1_06_1 (p[6][0],p[5][1],p[4][2],s1[06][1],c1[07][1]);
// index 06:   p[0][6]
add1 fa1_06_0 (p[3][3],p[2][4],p[1][5],s1[06][0],c1[07][0]);
add1 fa1_05_1 (p[5][0],p[4][1],p[3][2],s1[05][1],c1[06][1]);
add1 fa1_05_0 (p[2][3],p[1][4],p[0][5],s1[05][0],c1[06][0]);
add1 fa1_04_1 (p[4][0],p[3][1],p[2][2],s1[04][1],c1[05][1]);
add1 fa1_04_0 (p[1][3],p[0][4],zero,   s1[04][0],c1[05][0]);
add1 fa1_03_0 (p[3][0],p[2][1],p[1][2],s1[03][0],c1[04][0]);
// index 03:   p[0][3]
add1 fa1_02_0 (p[2][0],p[1][1],p[0][2],s1[02][0],c1[03][0]);
add1 fa1_01_0 (p[1][0],p[0][1],zero,   s1[01][0],c1[02][0]);
assign z[1] = s1[01][0];
```

以下是第 2 级加法器的代码。

```
wire [1:0] s2[13:02];
wire [1:0] c2[14:03];
// index 14:   p[7][7]
add1 fa2_13_0 ( p[7][6],  p[6][7], c1[13][0],s2[13][0],c2[14][0]);
add1 fa2_12_0 (s1[12][0],c1[12][0],zero,     s2[12][0],c2[13][0]);
add1 fa2_11_0 (s1[11][0], p[4][7], c1[11][0],s2[11][0],c2[12][0]);
// index 11:  c1[11][0]
add1 fa2_10_0 (s1[10][1],s1[10][0],c1[10][0],s2[10][0],c2[11][0]);
// index 10:  c1[10][0]
add1 fa2_09_0 (s1[09][1],s1[09][0],c1[09][1],s2[09][0],c2[10][0]);
// index 09:  c1[09][0]
add1 fa2_08_1 (s1[08][1],s1[08][0], p[1][7], s2[08][1],c2[09][1]);
add1 fa2_08_0 (c1[08][2],c1[08][1],c1[08][0],s2[08][0],c2[09][0]);
add1 fa2_07_1 (s1[07][2],s1[07][1],s1[07][0],s2[07][1],c2[08][1]);
```

```
add1 fa2_07_0 (c1[07][1],c1[07][0],zero,     s2[07][0],c2[08][0]);
add1 fa2_06_1 (s1[06][1],s1[06][0], p[0][6], s2[06][1],c2[07][1]);
add1 fa2_06_0 (c1[06][1],c1[06][0],zero,     s2[06][0],c2[07][0]);
add1 fa2_05_0 (s1[05][1],s1[05][0],c1[05][1],s2[05][0],c2[06][0]);
// index 05:   c1[05][0]
add1 fa2_04_0 (s1[04][1],s1[04][0],c1[04][0],s2[04][0],c2[05][0]);
add1 fa2_03_0 (s1[03][0], p[0][3], c1[03][0],s2[03][0],c2[04][0]);
add1 fa2_02_0 (s1[02][0],c1[02][0],zero,     s2[02][0],c2[03][0]);
assign z[2] = s2[02][0];
```

以下是第 3 级加法器的代码。

```
wire [11:03] s3;
wire [12:04] c3;
// index 14:  p[7][7], c2[14][0]
// index 13: s2[13][0],c2[13][0]
// index 12: s2[12][0],c2[12][0]
add1 fa3_11_0 (s2[11][0],c1[11][0],c2[11][0],s3[11],c3[12]);
add1 fa3_10_0 (s2[10][0],c1[10][0],c2[10][0],s3[10],c3[11]);
add1 fa3_09_0 (s2[09][0],c1[09][0],c2[09][1],s3[09],c3[10]);
// index 09:   c2[09][0]
add1 fa3_08_0 (s2[08][1],s2[08][0],c2[08][0],s3[08],c3[09]);
add1 fa3_07_0 (s2[07][1],s2[07][0],c2[07][1],s3[07],c3[08]);
// index 07:   c2[07][0]
add1 fa3_06_0 (s2[06][1],s2[06][0],c2[06][0],s3[06],c3[07]);
add1 fa3_05_0 (s2[05][0],c1[05][0],c2[05][0],s3[05],c3[06]);
add1 fa3_04_0 (s2[04][0],c2[04][0],zero,     s3[04],c3[05]);
add1 fa3_03_0 (s2[03][0],c2[03][0],zero,     s3[03],c3[04]);
assign z[3] = s3[03];
```

以下是第 4 级加法器的代码。最后使用一般的进位传播加法器加 s4 和 c4，得到结果 z 的高位部分。进位传播加法器可以是先行进位加法器。

```
wire [14:04] s4;
wire [15:05] c4;
add1 fa4_14_0 ( p[7][7], c2[14][0],zero,  s4[14],c4[15]);
add1 fa4_13_0 (s2[13][0],c2[13][0],zero,  s4[13],c4[14]);
add1 fa4_12_0 (s2[12][0],c2[12][0],c3[12],s4[12],c4[13]);
add1 fa4_11_0 (s3[11],   c3[11],   zero,  s4[11],c4[12]);
add1 fa4_10_0 (s3[10],   c3[10],   zero,  s4[10],c4[11]);
add1 fa4_09_0 (s3[09],   c2[09][0],c3[09],s4[09],c4[10]);
add1 fa4_08_0 (s3[08],   c2[08][0],c3[08],s4[08],c4[09]);
add1 fa4_07_0 (s3[07],   c2[07][0],c3[07],s4[07],c4[08]);
add1 fa4_06_0 (s3[06],   c3[06],   zero,  s4[06],c4[07]);
add1 fa4_05_0 (s3[05],   c3[05],   zero,  s4[05],c4[06]);
add1 fa4_04_0 (s3[04],   c3[04],   zero,  s4[04],c4[05]);
assign z[4] = s4[04];
```

```
    assign z[15:5] = {1'b0,s4[14:05]} + c4[15:05];
endmodule
```

图 3.18 给出 8 × 8 Wallace 树型乘法器的仿真结果。

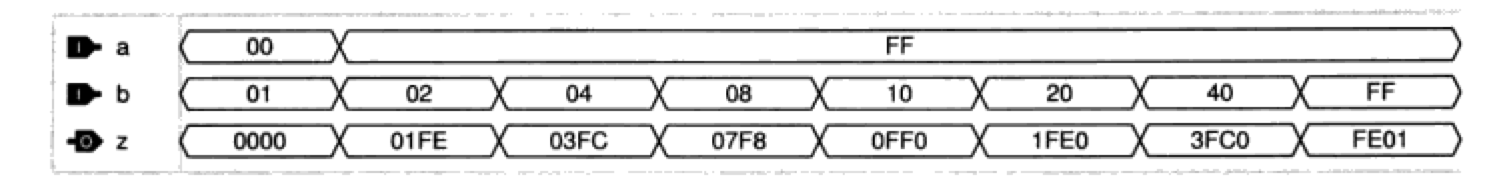


图 3.18　8 × 8 Wallace 树型乘法器仿真结果

3.3.4 带符号数 Wallace 树型乘法器设计

我们仍使用已经描述过的带符号数乘法算法来设计带符号数 Wallace 树型乘法器。以下是带符号数的 8 × 8 Wallace 树型乘法器的 Verilog HDL 代码。与无符号数的 Wallace 树型乘法器的代码相比，有以下两点不同：(1) 乘积项的产生不仅用了与门，而且也用了与非门；(2) 由于要加 1000 0001 0000 0000，第 08 位的第 1 级输入变为 8 个，因此需要 3 个全加器。

```
module wallace_tree8_signed (a,b,z);
    input  [7:0] a,b;
    output [15:0] z;
    reg [7:0] p[7:0];
    always @ * begin
        integer i,j;
        for (i = 0; i < 7; i = i + 1)
            for (j =  0; j < 7; j = j + 1)
                p[i][j] = a[i] & b[j]; //  AND
        for (i = 0; i < 7; i = i + 1)
            p[i][7] = ~(a[i] & b[7]);  // NAND
        for (j = 0; j < 7; j = j + 1)
            p[7][j] = ~(a[7] & b[j]);  // NAND
        p[7][7] = a[7] & b[7];         //  AND
    end
    assign z[0] = p[0][0];
    parameter zero = 1'b0;
    parameter one  = 1'b1; // + 1000_0001_0000_0000
    wire [2:0] s1[12:01];
    wire [2:0] c1[13:02];
    add1 fa1_12_0 (p[7][5],p[6][6],p[5][7],s1[12][0],c1[13][0]);
    add1 fa1_11_0 (p[7][4],p[6][5],p[5][6],s1[11][0],c1[12][0]);
    add1 fa1_10_1 (p[7][3],p[6][4],p[5][5],s1[10][1],c1[11][1]);
    add1 fa1_10_0 (p[4][6],p[3][7],zero,   s1[10][0],c1[11][0]);
    add1 fa1_09_1 (p[7][2],p[6][3],p[5][4],s1[09][1],c1[10][1]);
    add1 fa1_09_0 (p[4][5],p[3][6],p[2][7],s1[09][0],c1[10][0]);
    add1 fa1_08_2 (p[7][1],p[6][2],p[5][3],s1[08][2],c1[09][2]);
```

```
add1 fa1_08_1 (p[4][4],p[3][5],p[2][6],s1[08][1],c1[09][1]);
add1 fa1_08_0 (p[1][7],one,    zero,    s1[08][0],c1[09][0]); // 1
add1 fa1_07_2 (p[7][0],p[6][1],p[5][2],s1[07][2],c1[08][2]);
add1 fa1_07_1 (p[4][3],p[3][4],p[2][5],s1[07][1],c1[08][1]);
add1 fa1_07_0 (p[1][6],p[0][7],zero,    s1[07][0],c1[08][0]);
add1 fa1_06_1 (p[6][0],p[5][1],p[4][2],s1[06][1],c1[07][1]);
add1 fa1_06_0 (p[3][3],p[2][4],p[1][5],s1[06][0],c1[07][0]);
add1 fa1_05_1 (p[5][0],p[4][1],p[3][2],s1[05][1],c1[06][1]);
add1 fa1_05_0 (p[2][3],p[1][4],p[0][5],s1[05][0],c1[06][0]);
add1 fa1_04_1 (p[4][0],p[3][1],p[2][2],s1[04][1],c1[05][1]);
add1 fa1_04_0 (p[1][3],p[0][4],zero,    s1[04][0],c1[05][0]);
add1 fa1_03_0 (p[3][0],p[2][1],p[1][2],s1[03][0],c1[04][0]);
add1 fa1_02_0 (p[2][0],p[1][1],p[0][2],s1[02][0],c1[03][0]);
add1 fa1_01_0 (p[1][0],p[0][1],zero,    s1[01][0],c1[02][0]);
assign z[1] = s1[01][0];
wire [1:0] s2[13:02];
wire [1:0] c2[14:03];
add1 fa2_13_0 ( p[7][6],  p[6][7], c1[13][0],s2[13][0],c2[14][0]);
add1 fa2_12_0 (s1[12][0],c1[12][0],zero,     s2[12][0],c2[13][0]);
add1 fa2_11_0 (s1[11][0], p[4][7], c1[11][1],s2[11][0],c2[12][0]);
add1 fa2_10_0 (s1[10][1],s1[10][0],c1[10][1],s2[10][0],c2[11][0]);
add1 fa2_09_1 (s1[09][1],s1[09][0],c1[09][2],s2[09][1],c2[10][1]);
add1 fa2_09_0 (c1[09][1],c1[09][0],zero,     s2[09][0],c2[10][0]);
add1 fa2_08_1 (s1[08][2],s1[08][1],s1[08][0],s2[08][1],c2[09][1]);
add1 fa2_08_0 (c1[08][2],c1[08][1],c1[08][0],s2[08][0],c2[09][0]);
add1 fa2_07_1 (s1[07][2],s1[07][1],s1[07][0],s2[07][1],c2[08][1]);
add1 fa2_07_0 (c1[07][1],c1[07][0],zero,     s2[07][0],c2[08][0]);
add1 fa2_06_1 (s1[06][1],s1[06][0], p[0][6], s2[06][1],c2[07][1]);
add1 fa2_06_0 (c1[06][1],c1[06][0],zero,     s2[06][0],c2[07][0]);
add1 fa2_05_0 (s1[05][1],s1[05][0],c1[05][1],s2[05][0],c2[06][0]);
add1 fa2_04_0 (s1[04][1],s1[04][0],c1[04][0],s2[04][0],c2[05][0]);
add1 fa2_03_0 (s1[03][0], p[0][3], c1[03][0],s2[03][0],c2[04][0]);
add1 fa2_02_0 (s1[02][0],c1[02][0],zero,     s2[02][0],c2[03][0]);
assign z[2] = s2[02][0];
wire [11:03] s3;
wire [12:04] c3;
add1 fa3_11_0 (s2[11][0],c1[11][0],c2[11][0],s3[11],c3[12]);
add1 fa3_10_0 (s2[10][0],c1[10][0],c2[10][1],s3[10],c3[11]);
add1 fa3_09_0 (s2[09][1],s2[09][0],c2[09][1],s3[09],c3[10]);
add1 fa3_08_0 (s2[08][1],s2[08][0],c2[08][1],s3[08],c3[09]);
add1 fa3_07_0 (s2[07][1],s2[07][0],c2[07][1],s3[07],c3[08]);
add1 fa3_06_0 (s2[06][1],s2[06][0],c2[06][0],s3[06],c3[07]);
add1 fa3_05_0 (s2[05][0],c1[05][0],c2[05][0],s3[05],c3[06]);
add1 fa3_04_0 (s2[04][0],c2[04][0],zero,     s3[04],c3[05]);
add1 fa3_03_0 (s2[03][0],c2[03][0],zero,     s3[03],c3[04]);
assign z[3] = s3[03];
wire [14:04] s4;
```

```
    wire [15:05] c4;
    add1 fa4_14_0 ( p[7][7], c2[14][0],zero,  s4[14],c4[15]);
    add1 fa4_13_0 (s2[13][0],c2[13][0],zero,  s4[13],c4[14]);
    add1 fa4_12_0 (s2[12][0],c2[12][0],c3[12],s4[12],c4[13]);
    add1 fa4_11_0 (s3[11],   c3[11],   zero,  s4[11],c4[12]);
    add1 fa4_10_0 (s3[10],   c2[10][0],c3[10],s4[10],c4[11]);
    add1 fa4_09_0 (s3[09],   c2[09][0],c3[09],s4[09],c4[10]);
    add1 fa4_08_0 (s3[08],   c2[08][0],c3[08],s4[08],c4[09]);
    add1 fa4_07_0 (s3[07],   c2[07][0],c3[07],s4[07],c4[08]);
    add1 fa4_06_0 (s3[06],   c3[06],   zero,  s4[06],c4[07]);
    add1 fa4_05_0 (s3[05],   c3[05],   zero,  s4[05],c4[06]);
    add1 fa4_04_0 (s3[04],   c3[04],   zero,  s4[04],c4[05]);
    assign z[4] = s4[04];
    assign z[15:5] = {one,s4[14:05]} + c4[15:05]; // 1
endmodule
```

仿真结果与图 3.14 (带符号数相乘) 相同。

3.4 除法算法及 Verilog HDL 实现

本节介绍各种除法算法，包括无符号数的“恢复余数”和“不恢复余数”除法算法、带符号数的不恢复余数除法算法、Goldschmidt 算法以及 Newton-Raphson 算法，并给出所有这些算法的 Verilog HDL 代码。

3.4.1 恢复余数除法器设计

恢复余数除法的基本思路是从“部分余数”中减去除数，如果结果为负 (不够减)，则恢复原来的部分余数，商 0。我们使用 3 个寄存器：reg_q、reg_b 和 reg_r。开始时，reg_q 存放被除数 a，reg_b 存放除数 b，reg_r 清零。计算完成后，reg_q 中的内容是商，reg_r 中的内容是余数。图 3.19 所示的是 32 位除以 16 位的电路图。

做减法时，减数是 reg_b 中的内容 (除数)，被减数是 reg_r 的内容 (余数) 左移一位，最低位由 reg_q (被除数) 的最高位补充。为了能够判断相减结果的正负，减法器的位数要比除数的位数多出一位。如果相减结果 (部分余数) 为正 (减法器输出的最高位是 0)，商 1 (非门的输出)，把相减结果写入 reg_r (多路器选右端的输入)，reg_q 的内容左移一位，最低位放入商 1。如果相减结果为负，商 0，把被减数写入 reg_r (多路器选左端的输入，相当于恢复余数)，reg_q 的内容左移一位，最低位放入商 0。如此循环往复，直到被除数全被移出 reg_q 为止。

恢复余数的任务由 reg_r 上面的多路器完成，多路器的选择信号是相减结果的符号位。被减数的左移操作通过适当的连线实现，被除数的左移操作由右边的二选一多路器经适当的连线完成。为什么还要用一个多路器？因为开始时要把被除数打入寄存器 reg_q，迭代时要把 reg_q 的内容左移。

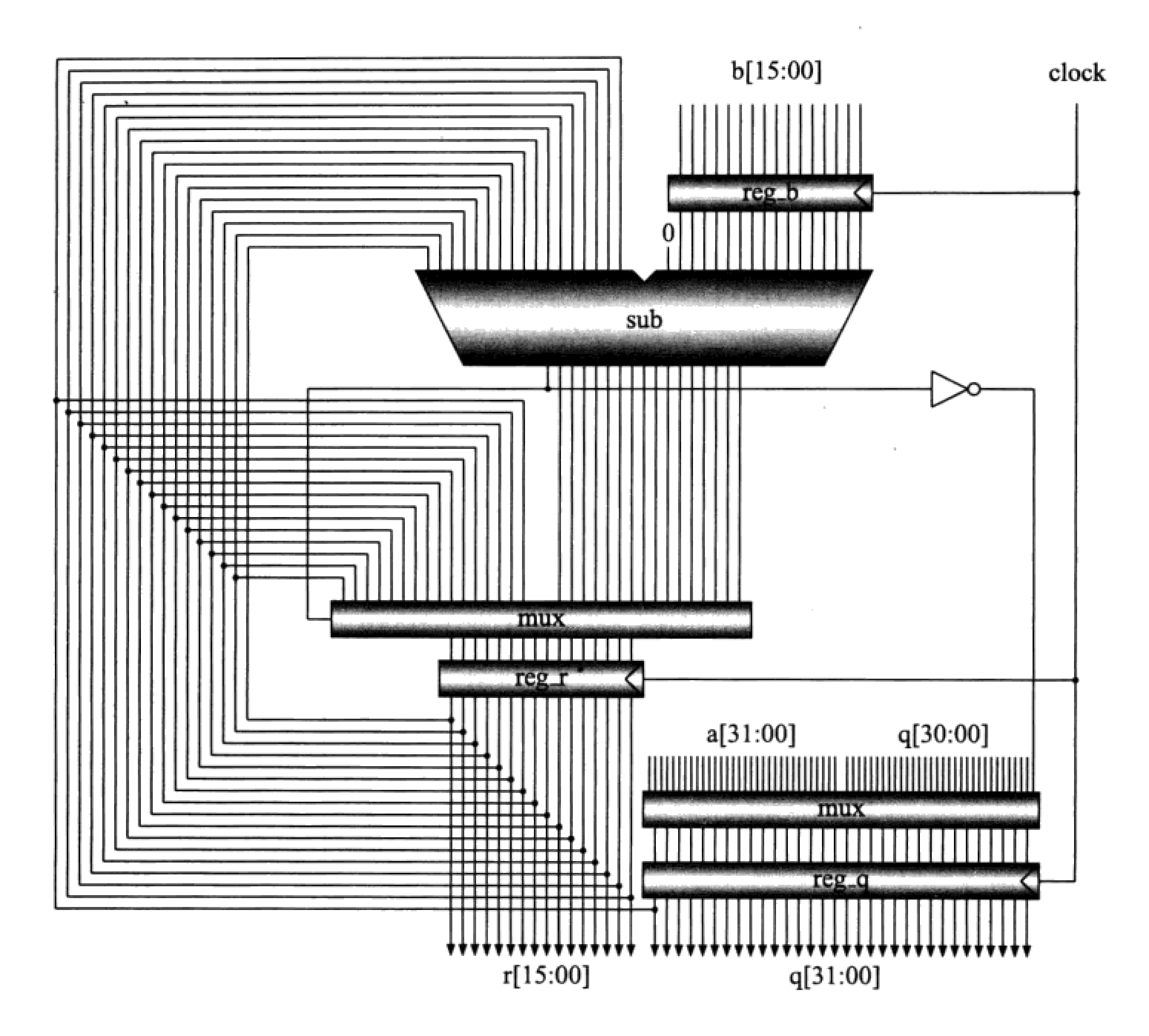


图 3.19 恢复余数除法器总体电路图

恢复余数除法器的 Verilog HDL 代码如下。输入信号 a 是被除数，b 是除数。输出信号 q 是商，r 是余数。另外几个重要的信号需要解释一下：start 表示启动除法运算；busy 表示除法器忙着呢；ready 表示结果出来了；count 是一个计数器，控制迭代次数。

```
module div_restoring (a,b,start,clock,resetn,q,r,busy,ready,count);
    input [31:00] a;     // dividend
    input [15:00] b;     // divisor
    input start;         // ID stage: start = is_div & ~busy;
    input clock,resetn;
    output [31:00] q;    // quotient
    output [15:00] r;    // remainder
    output busy;         // cannot receive new div
    output ready;        // ready to save result
    output [4:0] count;  // for sim test only
    reg [31:00] reg_q;
    reg [15:00] reg_r;
```

```
    reg [15:00] reg_b;
    reg [4:0] count;
    reg busy,busy2;
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count <= 5'b0;      // reset count
            busy  <= 0;         // reset to not busy
            busy2 <= 0;         // for generating 1-cycle ready
        end else begin          // not reset
            busy2 <= busy;      // 1-cycle delay of busy
            if (start) begin    // start: 1 cycle only
                reg_r <= 16'h0; // reset remainder
                reg_q <= a;     // load a
                reg_b <= b;     // load b
                count <= 5'b0;  // reset count
                busy  <= 1'b1;  // set to busy
             end else if (busy) begin  // execution: 32 cycles
                reg_r <= mux_out;      // partial remainder
                reg_q <= {reg_q[30:00],~sub_out[16]}; // 1-bit q
                count <= count + 5'b1; // count++
                if (count == 5'h1f) busy <= 0; // finish
            end
        end
    end
    assign ready = ~busy & busy2; // generate 1-cycle ready
    wire [16:00] sub_out = {r,q[31]} - {1'b0,reg_b}; // sub
    wire [15:00] mux_out = sub_out[16]?  // restoring or not
                          {r[14:0],q[31]} : sub_out[15:00];
    assign q = reg_q;
    assign r = reg_r;
endmodule
```

恢复余数除法器的仿真结果如图 3.20 所示，其中的计数器只做参考用。

3.4.2 不恢复余数除法器设计

在恢复余数除法算法中，如果部分余数为负，则要恢复原来的余数并左移。设部分余数为 R，除数为 B。恢复余数相当于 $R + B$，左移相当于 $(R + B) \times 2$。以上操作完成后进行下一轮的迭代，即从部分余数中减去 B。我们有以下的等式：

$$(R + B) \times 2 - B = R \times 2 + B$$

这就是不恢复余数除法算法的中心思想。即，不管相减结果是正是负，都把它写入 reg_r，若为负，下次迭代不是从中减去除数而是加上除数。图 3.21 是不恢复余数除法器总体电路图。图中左下角的加法器和多路器是为了得到正确的余数而设置的，如果在你的除法电路中不需要余数的话，可以扔掉它们。

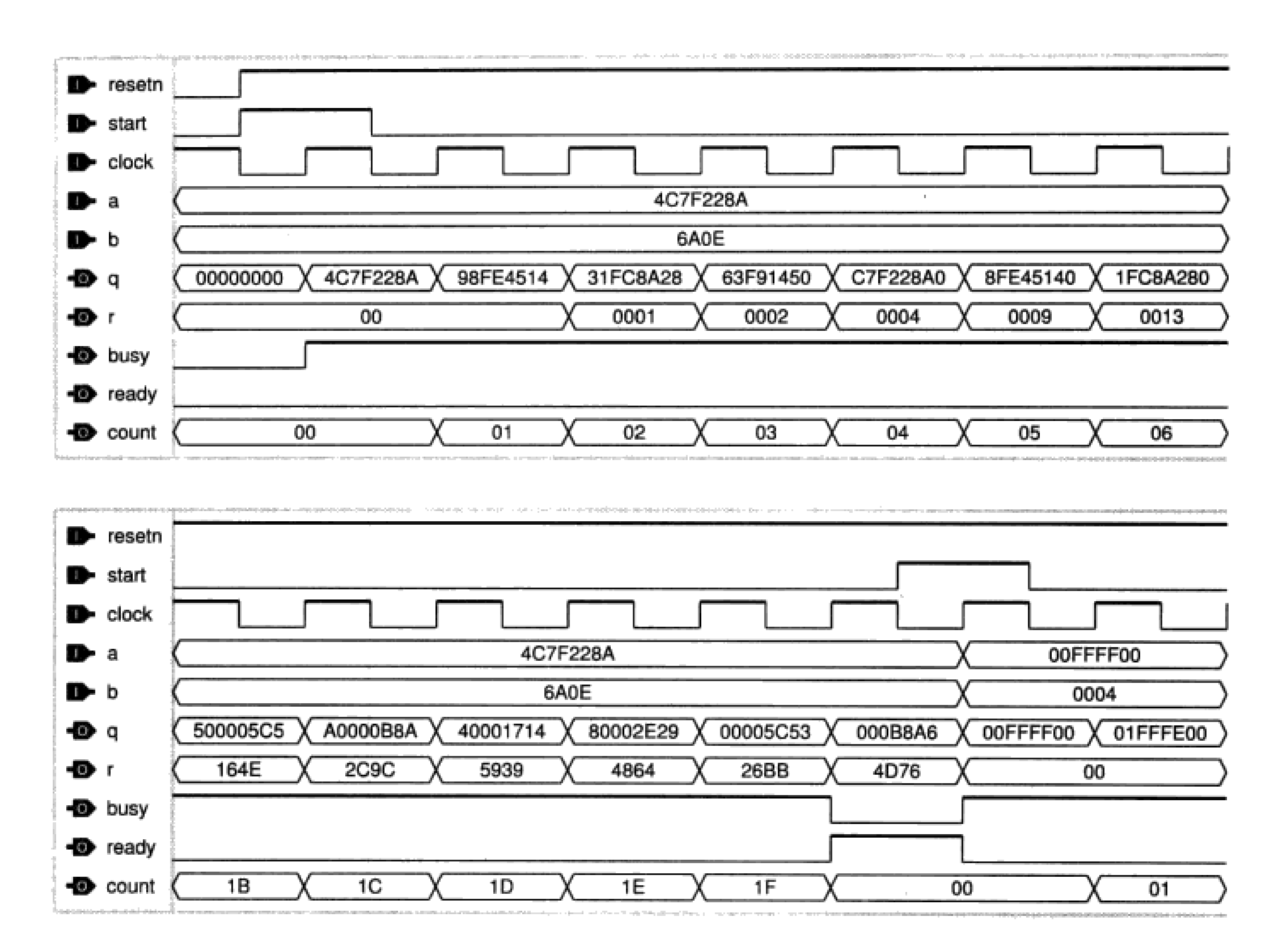


图 3.20　恢复余数除法器仿真结果

不恢复余数除法器的 Verilog HDL 代码如下。

```
module div_nonrestoring (a, b, start, clock, resetn, q, r, busy,
                         ready, count);
    input  [31:00] a;     // dividend
    input  [15:00] b;     // divisor
    input  start;         // ID stage: start = is_div & ~busy;
    input  clock,resetn;
    output [31:00] q;     // quotient
    output [15:00] r;     // remainder
    output busy;          // cannot receive new div
    output ready;         // ready to save result
    output [4:0] count;   // for sim test only
    reg [31:00] reg_q;
    reg [15:00] reg_r;
    reg [15:00] reg_b;
    reg [4:0] count;
    reg busy,busy2,r_sign;
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count <= 5'b0;         // reset count
            busy  <= 0;            // reset to not busy
            busy2 <= 0;            // for generating 1-cycle ready
```

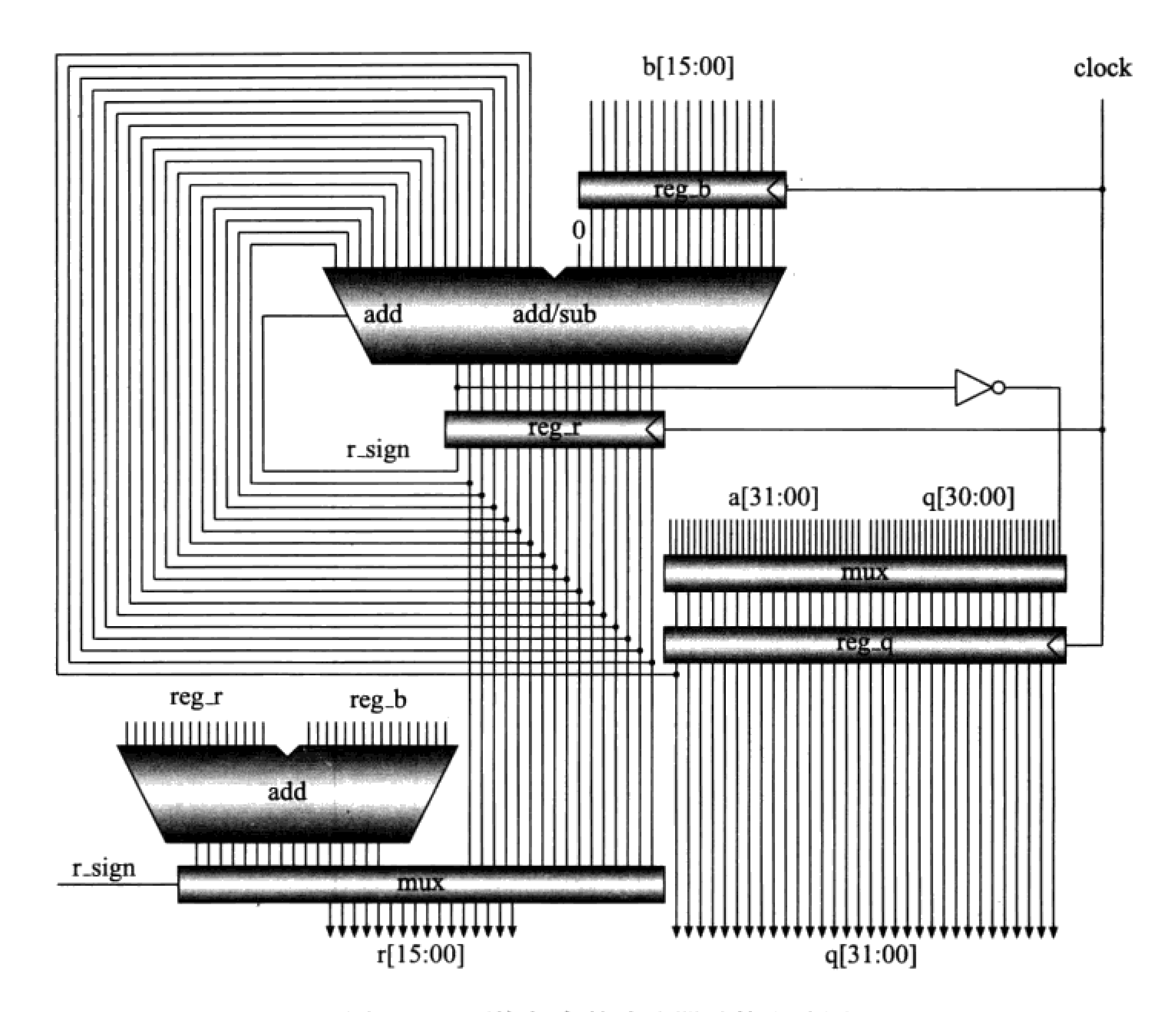


图 3.21 不恢复余数除法器总体电路图

```
    end else begin              // not reset
        busy2 <= busy;          // 1-cycle delay of busy
        if (start) begin        // start: 1 cycle only
            reg_r  <= 16'h0;    // reset remainder
            r_sign <= 0;        // sub first
            reg_q  <= a;        // load a
            reg_b  <= b;        // load b
            count  <= 5'b0;     // reset count
            busy   <= 1'b1;     // set to busy
         end else if (busy) begin      // execution: 32 cycles
            reg_r  <= sub_add[15:00]; // partial remainder
            r_sign <= sub_add[16];    // if minus, add next
            reg_q  <= {reg_q[30:00],~sub_add[16]}; // 1-bit q
            count  <= count + 5'b1;   // count++
            if (count == 5'h1f) busy <= 0; // finish
        end
    end
end
```

```
    assign ready = ~busy & busy2; // generate 1-cycle ready
    wire [16:00] sub_add = r_sign? {reg_r,q[31]} + {1'b0,reg_b} :
                                   {reg_r,q[31]} - {1'b0,reg_b};
    assign r = r_sign? reg_r + reg_b : reg_r; // adjust remainder
    assign q = reg_q;
endmodule
```

不恢复余数除法器 Verilog HDL 代码的仿真结果如图 3.22 所示。

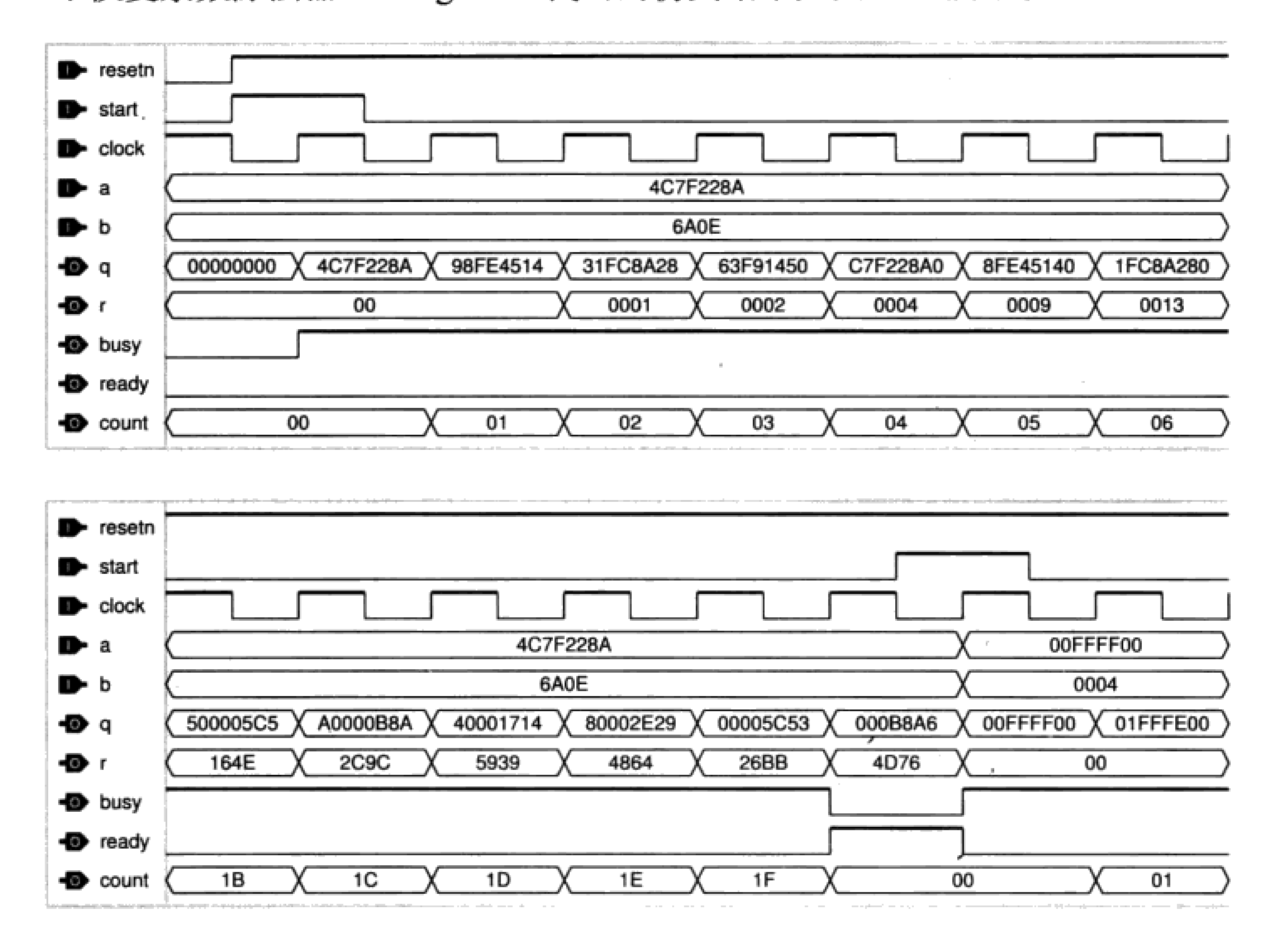


图 3.22 不恢复余数除法器仿真结果

3.4.3 带符号数不恢复余数除法器设计

带符号数的除法要考虑被除数和除数的符号。如果两个数的符号相同，商为正，否则为负。带符号数除法最直接的算法是首先把它们都转换成正数，然后按照无符号数的不恢复余数除法算法求出商的绝对值，然后再根据被除数和除数的符号对商进行调整。

本小节给出的算法略有不同。我们直接使用原始的被除数和除数进行计算，因为不恢复余数除法算法本身就允许部分余数为负数。在无符号数的不恢复余数除法算法中，如果部分余数为负，下次计算要加上除数。现将该规则改为：如果部分余数 (第一次为被除数) 和除数的符号相同，则减去除数，否则加上除数。这样做的结果导致求出的商为正数，最后还要根据符号对商进行调整。图 3.23 所示的是带符号数不恢复余数除法器的总体电路图。

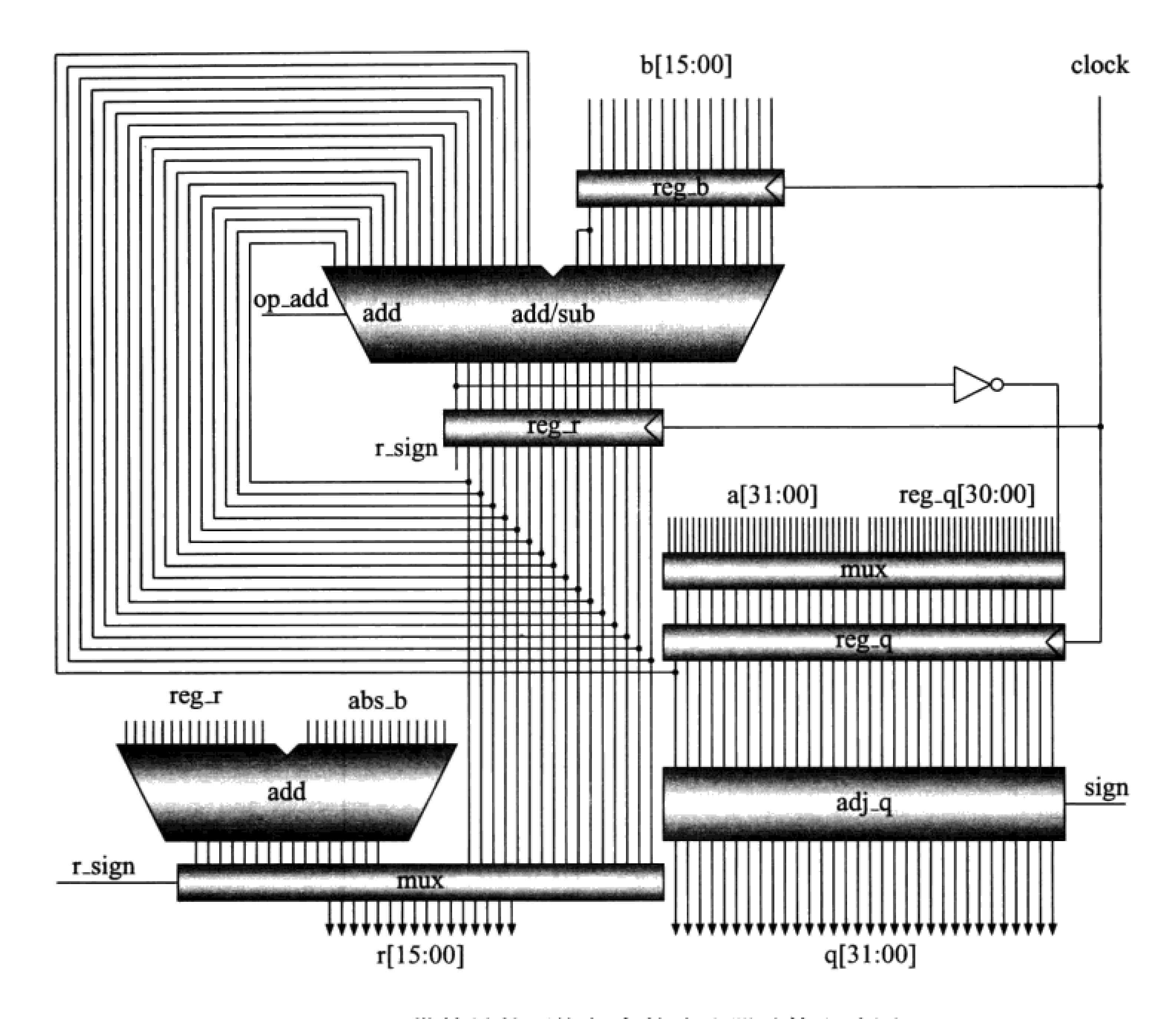


图 3.23 带符号数不恢复余数除法器总体电路图

带符号数不恢复余数除法器的 Verilog HDL 代码如下。

```
module div_nonrestoring_signed (a,b,start,clock,resetn,q,r,busy,
                                ready,count);
    input [31:00] a;     // dividend
    input [15:00] b;     // divisor
    input start;         // ID stage: start = is_div & ~busy;
    input clock,resetn;
    output [31:00] q;    // quotient
    output [15:00] r;    // remainder
    output busy;         // cannot receive new div
    output ready;        // ready to save result
    output [4:0] count; // for sim test only
    reg [31:00] reg_q;
    reg [15:00] reg_r;
    reg [15:00] reg_b;
    reg [4:0] count;
    reg busy,busy2,sign,r_sign;
```

```
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count <= 5'b0;    // reset count
            busy  <= 0;       // reset to not busy
            busy2 <= 0;       // for generating 1-cycle ready
        end else begin        // not reset
            busy2 <= busy;    // 1-cycle delay of busy
            if (start) begin // start: 1 cycle only
                reg_r  <= 16'h0;          // reset remainder
                r_sign <= a[31];          // if signs diff, add
                sign   <= a[31] ^ b[15]; // result sign
                reg_q  <= a;              // load a
                reg_b  <= b;              // load b
                count  <= 5'b0;           // reset count
                busy   <= 1'b1;           // set to busy
             end else if (busy) begin    // execution: 32 cycles
                reg_r  <= sub_add[15:00];       // partial remainder
                r_sign <= sub_add[16];          // if minus, add next
                reg_q[31:01] <= reg_q[30:00];  // shift left
                reg_q[00]    <= ~sub_add[16];  // 1-bit quotient
                count        <= count + 5'b1;  // count++
                if (count == 5'h1f) busy <= 0; // finish
            end
        end
    end
    assign ready = ~busy & busy2; // generate 1-cycle ready
    wire op_add = r_sign ^ reg_b[15]; // if signs diff, add
    wire [16:00] sub_add = op_add?
                         {reg_r,reg_q[31]} + {reg_b[15],reg_b} :
                         {reg_r,reg_q[31]} - {reg_b[15],reg_b};
    wire [15:00] abs_b = reg_b[15]? ~reg_b + 16'b1 : reg_b;
    assign r = r_sign? reg_r + abs_b : reg_r; // adjust remainder
    assign q = sign? ~reg_q + 1 : reg_q;      // adjust quotient
endmodule
```

带符号数不恢复余数除法器 Verilog HDL 代码的仿真结果如图 3.24 所示。

3.4.4 Goldschmidt 除法算法

设 a 和 b 均有 .1xxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx 的格式 (注意小数点)，即 $1/2 \leqslant a, b < 1$，以下介绍如何使用 Goldschmidt 算法计算 $q = a/b$。如果分式

$$\frac{a \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}}{b \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}}$$

的分母趋向于 1，则分子趋向于 q。定义 $\boldsymbol{\delta} = 1 - b$，则 $0 < \boldsymbol{\delta} \leqslant 1/2$，$b = 1 - \boldsymbol{\delta}$。

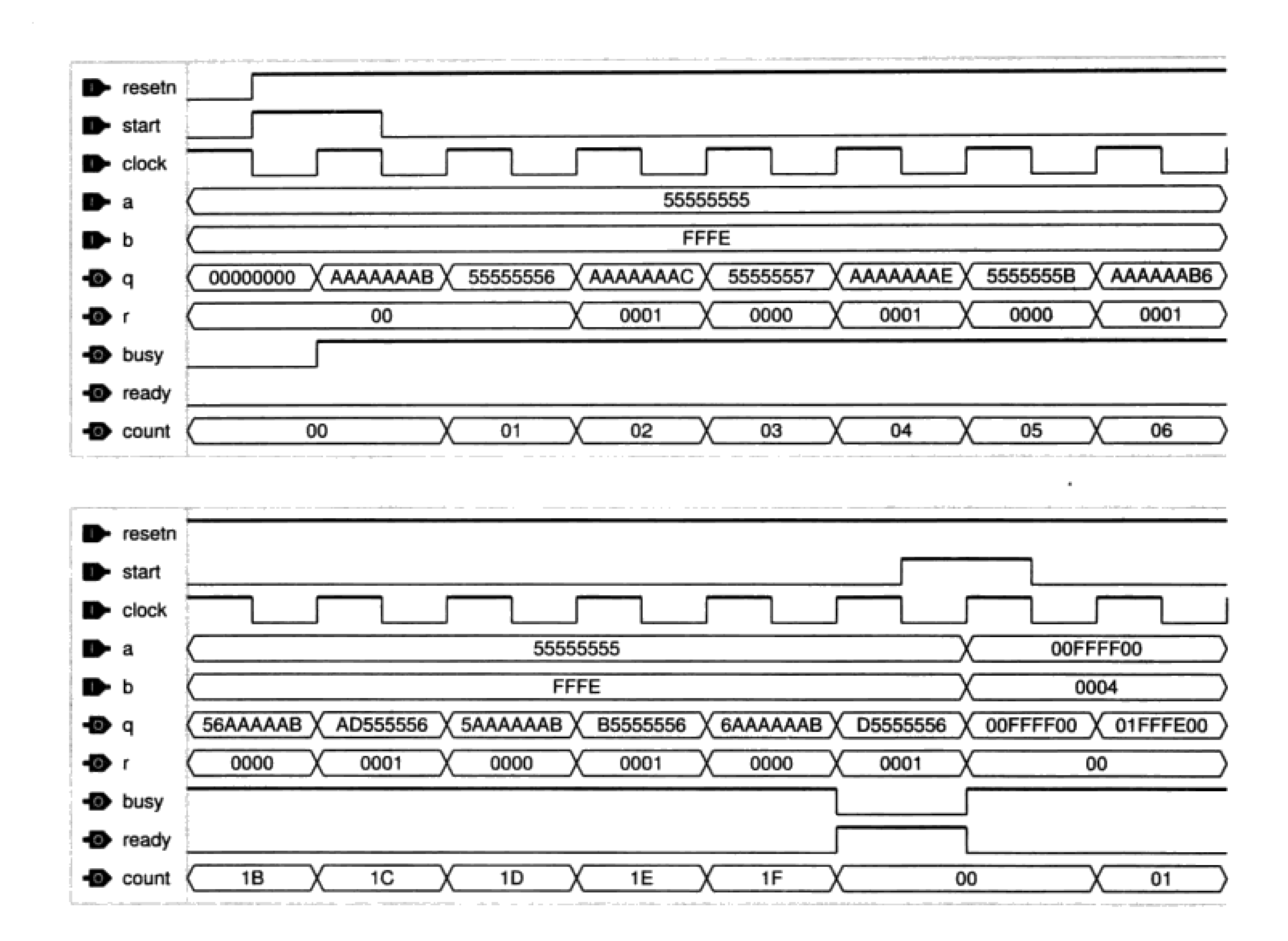


图 3.24 带符号数不恢复余数除法器仿真结果

定义 $x_0 = a$，$y_0 = b = 1 - \delta$，$r_0 = 2 - y_0 = 1 + \delta$。计算

$$
\begin{aligned}
x_1 &= x_0 \times r_0 \\
y_1 &= y_0 \times r_0 = (1-\delta) \times (1+\delta) = 1 - \delta^2 \\
r_1 &= 2 - y_1 = 1 + \delta^2 \\
x_2 &= x_1 \times r_1 \\
y_2 &= y_1 \times r_1 = (1-\delta^2) \times (1+\delta^2) = 1 - \delta^4 \\
r_2 &= 2 - y_2 = 1 + \delta^4 \\
&\cdots\cdots \\
x_{i+1} &= x_i \times r_i \\
y_{i+1} &= y_i \times r_i = (1-\delta^{2^i}) \times (1+\delta^{2^i}) = 1 - \delta^{2^{i+1}} \\
r_{i+1} &= 2 - y_{i+1} = 1 + \delta^{2^{i+1}}
\end{aligned}
$$

直到 y_{i+1} 接近于 1 为止。这时 x_{i+1} 接近于 q。为什么 $y_{i+1} \to 1$？这是因为 $y_{i+1} = 1 - \delta^{2^{i+1}}$ 而 $0 < \delta \leqslant 1/2$ 的原故。

Goldschmidt 除法器总体电路图如图 3.25 所示。两个寄存器 reg_a 和 reg_b 开始时分别存放被除数 a 和除数 b，迭代过程中分别存放 x_i 和 y_i。由于要满足 $1/2 \leqslant a, b < 1$，我们必须在该电路的外层对 a 和 b 进行预处理，对 q 进行善后处理。

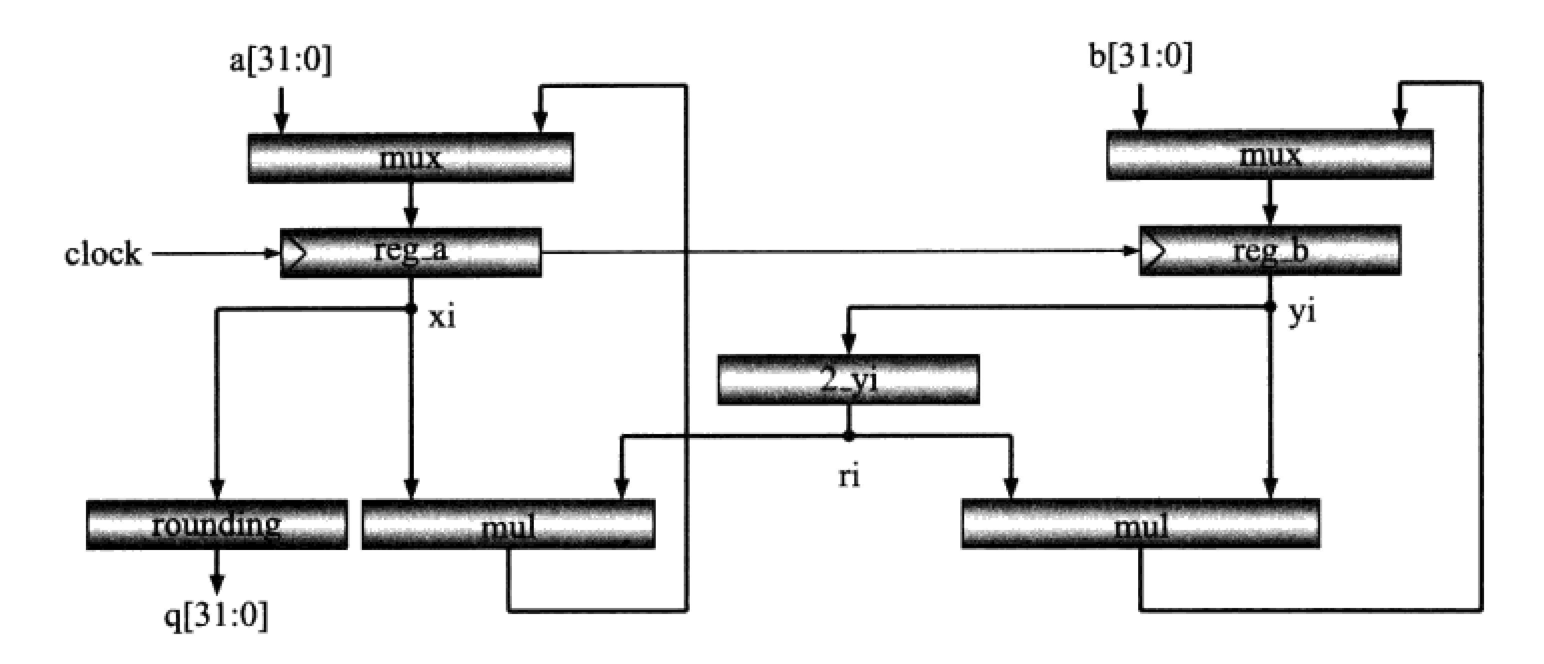


图 3.25 Goldschmidt 除法器总体电路图

以下是 Goldschmidt 除法器的 Verilog HDL 代码。注意小数点的位置。

```
module goldschmidt (a,b,start,clock,resetn,q,busy,ready,count,
                    reg_a,reg_b);
    input [31:00] a;      // dividend: fraction: .1xxx...x
    input [31:00] b;      // divisor:  fraction: .1xxx...x
    input start;          // ID_stage: start = is_div & ~busy;
    input clock,resetn;
    output [31:00] q;     // quotient: x.xxxx...x
    output busy;          // cannot receive new div
    output ready;         // ready to save result
    output [2:0] count;   // for sim test only
    output [33:00] reg_a; // for sim test only
    output [33:00] reg_b; // for sim test only
    reg [33:00] reg_a;    // 34-bit: x.xxxx...xx
    reg [33:00] reg_b;    // 34-bit: 0.1xxx...xx
    reg [2:0] count;      // 5 iterations
    reg busy,busy2;
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count <= 3'b0;    // reset count
            busy  <= 0;       // reset to not busy
            busy2 <= 0;       // for generating 1-cycle ready
        end else begin        // not reset
            busy2 <= busy;    // 1-cycle delay of busy
            if (start) begin // start: 1 cycle only
                reg_a <= {1'b0,a,1'b0}; // 0.1xxx...x0
                reg_b <= {1'b0,b,1'b0}; // 0.1xxx...x0
                count <= 3'b0;          // reset count
                busy  <= 1'b1;          // set to busy
             end else begin  // execution: 5 iterations
                reg_a <= a68[66:33];    // x.xxx...x
```

```
                reg_b <= b68[66:33];      // x.xxx...x
                count <= count + 3'b1;    // count++
                if (count == 3'h4) busy <= 0; // finish
            end
        end
    end
    wire [33:00] b34 = ~reg_b + 1'b1; // 2 - yi
    wire [67:00] a68 =  reg_a *  b34; // 0x.xxx...xx
    wire [67:00] b68 =  reg_b *  b34; // 0x.xxx...xx
    assign ready = ~busy & busy2; // generate 1-cycle ready
    assign q = reg_a[33:02] + (reg_a[01] | reg_a[00]); // rounding
endmodule
```

图 3.26 是 Goldschmidt 除法器仿真 $0.75/0.5 = 1.5$ 时的结果。被除数 0.75 用二进制数表示为 0.1100，其小数部分即是图中的输入信号 a。除数 0.5 用二进制数表示为 0.1000，其小数部分即是图中的输入信号 b。商 1.5 用二进制数表示为 1.100，是图中的输出信号 q。我们可以看出 b_i (图中的 reg_b) 逐渐趋向于 1。

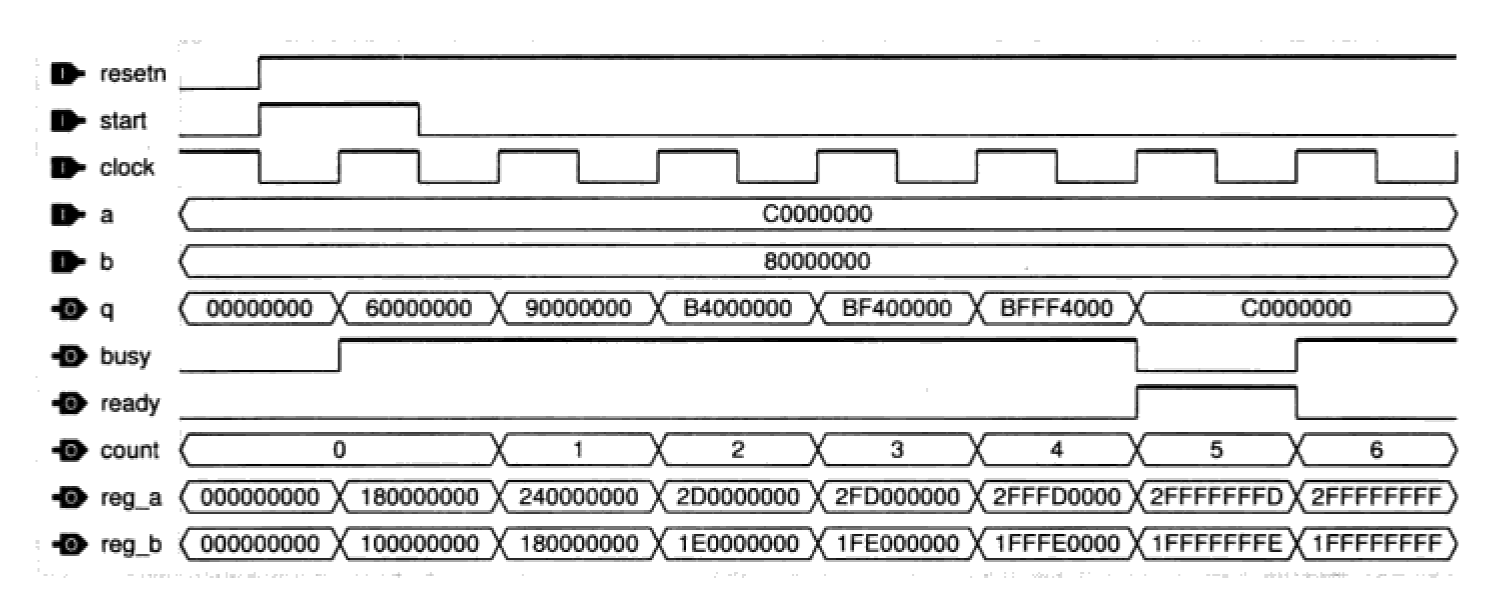


图 3.26 Goldschmidt 除法器仿真结果

迭代次数为 5 次，每次迭代包括一次乘法和一次加法。如果用 Wallace 树型乘法器，一次乘法可用两个加法周期完成。因此迭代部分需要 $5 \times 3 = 15$ 个周期。再加上最后的一个舍入周期，Goldschmidt 除法器总共需要 16 个周期。

3.4.5 Newton-Raphson 除法算法

Newton-Raphson 算法 (以下简称为牛顿迭代算法) 也使用乘法来代替除法运算。如果我们不用除法就能计算出 $1/b$ 的话，则 $a/b = a \times (1/b)$。假设我们有一个函数 $f(x)$，如何找出 x_n，使得 $f(x_n) \approx 0$? 首先，我们猜测 x_0，由 $f(x)$ 在该点处的切线方程 $y - f(x_0) = f'(x_0)(x - x_0)$，找出使 $y = 0$ 的新的一点 x_1，则 x_1 更加接近 x_n。

一般地，$y - f(x_i) = f'(x_i)(x - x_i)$，令 $y = 0$，我们有新的一点 $x_{i+1} = x_i - f(x_i)/f'(x_i)$。重复上述过程，一直到 x_n 足够精确为止。

令 $f(x) = 1/x - b$，则在 $x = 1/b$ 处 $f(x) = 0$。应用牛顿迭代公式，我们有:

$$x_{i+1} = x_i(2 - x_i b)$$

这样，我们能够用下述步骤实现 a/b:

1) 把 b 移位，使其满足 $0.5 \leqslant b < 1$，即，$b = 0.1xx\cdots xx_2$。
2) 使用 b 的高若干位查 ROM 表得到 1/b 的近似值 x_0，或者直接令 $x_0 = 1.5$。
3) 迭代 $x_{i+1} = x_i(2 - x_i b)$，直到 x_n 足够精确为止。
4) 计算 $a \times x_n$，把结果反向移位以消除第一步造成的影响。

设 x_i 精确到 p 位，这意味着 $|(x_i - 1/b)/(1/b)| \leqslant 2^{-p}$。通过推导，我们有 $|(x_{i+1} - 1/b)/(1/b)| \leqslant 2^{-2p}$，即，迭代一次精度是原来的两倍。

图 3.27 所示的是 Newton-Raphson 除法器总体电路图。图中的 3 个寄存器 reg_a、reg_b 和 reg_x 分别存放被除数 a、除数 b 和迭代变量 x_i。

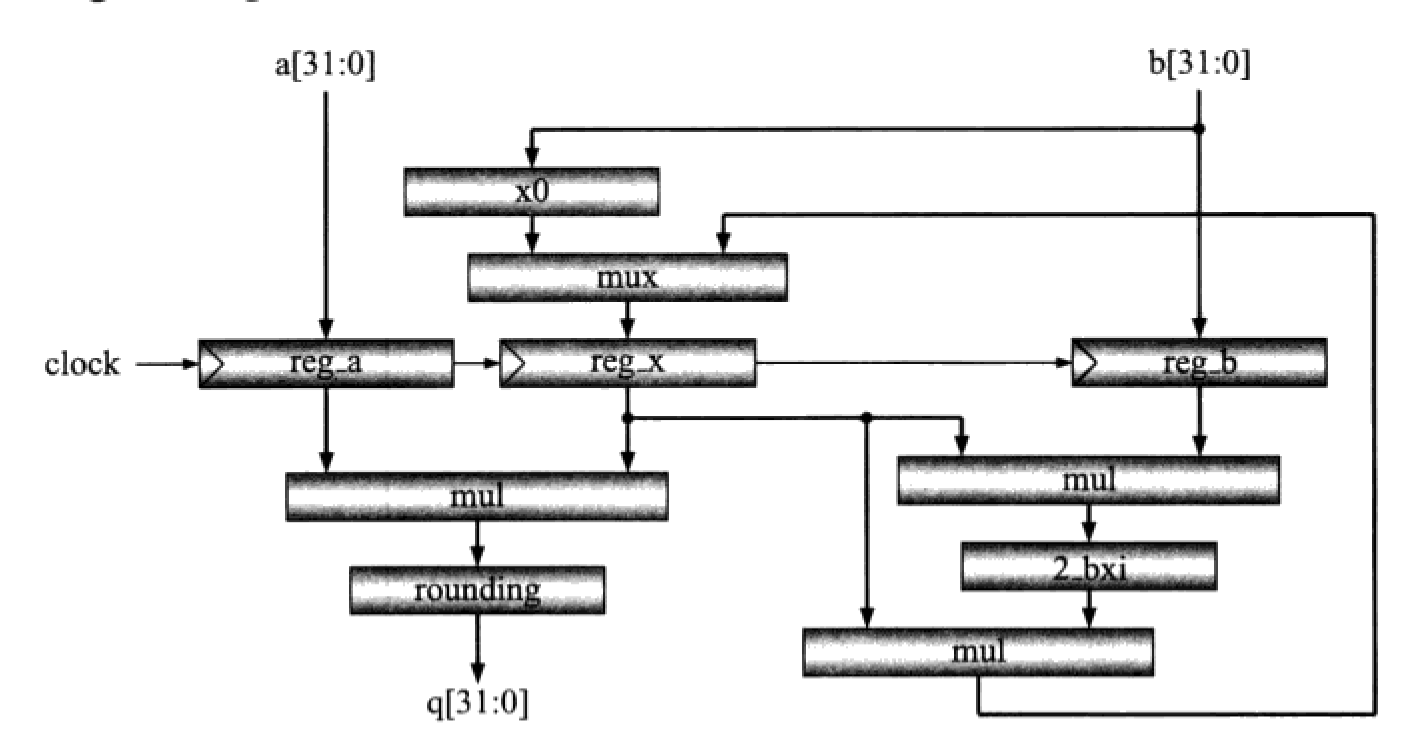


图 3.27 Newton-Raphson 除法器总体电路图

以下的 Verilog HDL 代码实现 Newton-Raphson 除法算法。注意小数点的位置。

```
module newton (a,b,start,clock,resetn,q,busy,ready,count,reg_x);
    input [31:00] a;       // dividend: fraction: .1xxx...x
    input [31:00] b;       // divisor:  fraction: .1xxx...x
    input start;           // ID stage: start = is_div & ~busy;
    input clock,resetn;
    output [31:00] q;      // a/b: x.xxxxx...x
    output busy;           // cannot receive new div
    output ready;          // ready to save result
    output [1:0] count;    // for sim test only
    output [33:00] reg_x; // for sim test only
    reg [33:00] reg_x;     // 34-bit: xx.xxxxx...xx
    reg [31:00] reg_a;     // 32-bit:   .1xxxx...xx
```

```
    reg [31:00] reg_b;    // 32-bit:   .1xxxx...xx
    reg [1:0] count;      // 3 iterations
    reg busy,busy2;
    // xi (2 - xi b)
    wire [7:0] x0 = rom(b[30:27]);
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count <= 2'b0;    // reset count
            busy  <= 0;       // reset to not busy
            busy2 <= 0;       // for generating 1-cycle ready
        end else begin        // not reset
            busy2 <= busy;    // 1-cycle delay of busy
            if (start) begin // start: 1 cycle only
                reg_x <= {2'b1,x0,24'b0};  // 01.xxxx0...0
                reg_a <= a;                //    .1xxxx...x
                reg_b <= b;                //    .1xxxx...x
                count <= 2'b0;             // reset count
                busy  <= 1'b1;             // set to busy
             end else begin  // execution: 3 iterations
                reg_x <= x68[66:33];       // xx.xxxxx...x
                count <= count + 2'b1;     // count++
                if (count == 2'h2) busy <= 0; // finish
            end
        end
    end
    wire [65:00] bxi =  reg_x * reg_b;     //  xx.xxxxx...x
    wire [33:00] b34 = ~bxi[64:31] + 1'b1; //   x.xxxxx...x
    wire [67:00] x68 =  reg_x * b34;       // xxx.xxxxx...x
    assign ready = ~busy & busy2; // generate 1-cycle ready
    wire [65:00] d_x = reg_a * reg_x;      //  xx.xxxxx...x
    assign q = d_x[64:33] + {31'h0,|d_x[32:0]}; // rounding
    function [7:0] rom;
        input [3:0] b;
        case (b)
            4'h0: rom = 8'hf0;             4'h1: rom = 8'hd4;
            4'h2: rom = 8'hba;             4'h3: rom = 8'ha4;
            4'h4: rom = 8'h8f;             4'h5: rom = 8'h7d;
            4'h6: rom = 8'h6c;             4'h7: rom = 8'h5c;
            4'h8: rom = 8'h4e;             4'h9: rom = 8'h41;
            4'ha: rom = 8'h35;             4'hb: rom = 8'h29;
            4'hc: rom = 8'h1f;             4'hd: rom = 8'h15;
            4'he: rom = 8'h0c;             4'hf: rom = 8'h04;
        endcase
    endfunction
endmodule
```

图 3.28 示出的是用 Newton-Raphson 除法器计算 0.75/0.5 = 1.5 时的仿真结果。被除数 0.75 用十六进制表示为 0.C0000000，其小数部分即是图中的输入信号 a。除数 0.5 用十六进制数表示为 0.80000000，其小数部分即是图中的输入信号 b。商 1.5 用二进制数表示为 1.100_0000_0000_0000_0000_0000_0000_0000，用十六进制表示为 C0000000，即图中的输出信号 q。我们可以看到 x_i (图中的 reg_x) 趋向于 2，即 1/b。a 与 1/b 相乘，得到 q (商)。

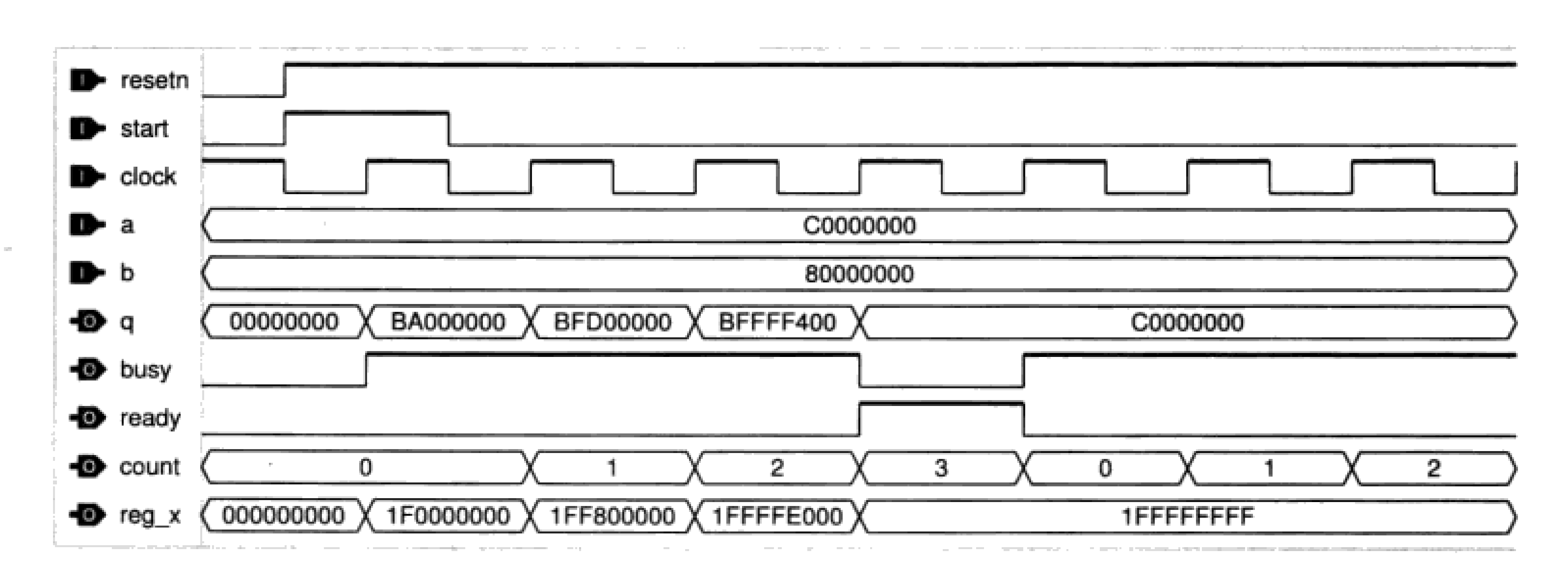


图 3.28 Newton-Raphson 除法器仿真结果

牛顿迭代一共做了 3 次，每次迭代包括两次乘法和一次加法。如果用 Wallace 树型乘法器，一次乘法可用两个加法周期完成。因此迭代部分需要 $3 \times 5 = 15$ 个周期。再加上开始的查表和最后的乘法及舍入，Newton-Raphson 除法器总共需要 $15 + 1 + 2 + 1 = 19$ 个周期。

3.5 开方算法及 Verilog HDL 实现

设有整数 d。开方运算可以归结为求 q 使 $q^2 + r = d$ 且 $r \leqslant 2q$。为什么需要条件 $r \leqslant 2q$ 呢？假设没有这个条件，则可永远令 $q = 1$、$r = d - 1$，这岂不是瞎耽误工夫吗？确实需要个条件，但为什么是 $r \leqslant 2q$ 呢？因为我们要使 q 是最大可能的整数，所以我们有 $q^2 + r = d < (q + 1)^2 = q^2 + 2q + 1$，即 $r < 2q + 1$。由于 d 和 q 都是整数，所以我们有 $r \leqslant 2q$。例如 $15 = 3^2 + 6$。

本节介绍几种开方求平方根的算法，包括“恢复余数”和“不恢复余数”开方算法、Goldschmidt 开方算法以及 Newton-Raphson 开方算法，并给出所有这些算法的 Verilog HDL 代码。

3.5.1 恢复余数开方算法

首先我们举例说明十进制数手算开方算法。设 $d = 3\,00\,00\,00\,00\,00$，试计算 $q = \sqrt{d}$。d 的两位对应 q 的一位，设 $q = q_1q_2q_3q_4q_5q_6$。以下是计算方法。

```
                        q1  q2  q3  q4  q5  q6
                         1   7   3   2   0   5    ; 平方根
        1             √03  00  00  00  00  00    ; 操作数
(q1 = 1) 1               1                        ; 1 × 1 = 1
                       ------------------------
        27               2 00                     ; 1 + 1 = 2
(q2 = 7)  7              1 89                     ; 27 × 7 = 189
                       ------------------------
        343                11 00                  ; 27 + 7 = 34
(q3 = 3)   3               10 29                  ; 343 × 3 = 1029
                           --------------------
        3462                  71 00               ; 343 + 3 = 346
(q4 = 2)    2                 69 24               ; 3462 × 2 = 6924
                              -----------------
        34640                  1 76 00            ; 3462 + 2 = 3464
(q5 = 0)     0                       0            ; 34640 × 0 = 0
                              -----------------
        346405                 1 76 00 00         ; 34640 + 0 = 34640
(q6 = 5)      5                1 73 20 25         ; 346405 × 5 = 1732025
                              -----------------
                                  2 79 75         ; 余数
```

二进制数开方也是一样，而且更简单。看例子：

```
                        q1  q2  q3  q4
                         1   0   1   1     ; 平方根 (11₁₀)
          1            √01  11  11  11     ; 操作数 (127₁₀)
(q1 = 1)  1              1                 ; 1 × 1 = 1
                       ----------------
          100            0 11              ; 1 + 1 = 10
(q2 = 0)   0               0               ; 100 × 0 = 0
                       ----------------
          1001             11 11           ; 100 + 0 = 100
(q3 = 1)     1             10 01           ; 1001 × 1 = 1001
                           ------------
          10101             1 10 11        ; 1001 + 1 = 1010
(q4 = 1)      1             1 01 01        ; 10101 × 1 = 10101
                           ------------
                              01 10        ; 余数 (6₁₀)
```

恢复余数开方算法的基本思路是从部分余数 (开始时是 d 的最高两位，设 d 有偶数位) 减去 Q01 (即 Q << 2 + 1)。Q 是部分平方根，开始设为 0。如果够减，得到一位平方根 1，把 Q 左移一位后加 1，部分余数是相减结果左移两位再拼接上 d 的高两位 (d 也要左移两位)。如果不够减，把 Q 左移一位后加 0，部分余数是被减数左移两位再拼接上 d 的高两位。

图 3.29 是恢复余数开方电路总体图。寄存器 reg_d 开始时存放 d，迭代时存放左移后的 d (每次左移两位)，因此该寄存器的输入端需要一个多路器。寄存器 reg_q 开始时清零，迭代时存放部分平方根 (左移一位)，迭代结束时的内容是平方根 q。寄存器 reg_r 开始时也清零，迭代过程中保存部分余数，迭代结束时的内容是余数 r。

每次迭代时，把部分余数 (reg_r 中的内容) 左移两位与 reg_d 中的最高两位拼起来，作为被减数使用；同时，把部分平方根 (reg_q 中的内容) 左移两位与 01 拼起来，作为减数使用。相减结果若非负 (加法器输出的最高位为 0)，得到一位平方根 1，把相减结果存入寄存器 reg_r；相减结果若为负 (加法器输出的最高位为 1)，得到一位平方根 0，把被减数存入寄存器 reg_r。此处即为“恢复余数”的意义之所在。寄存

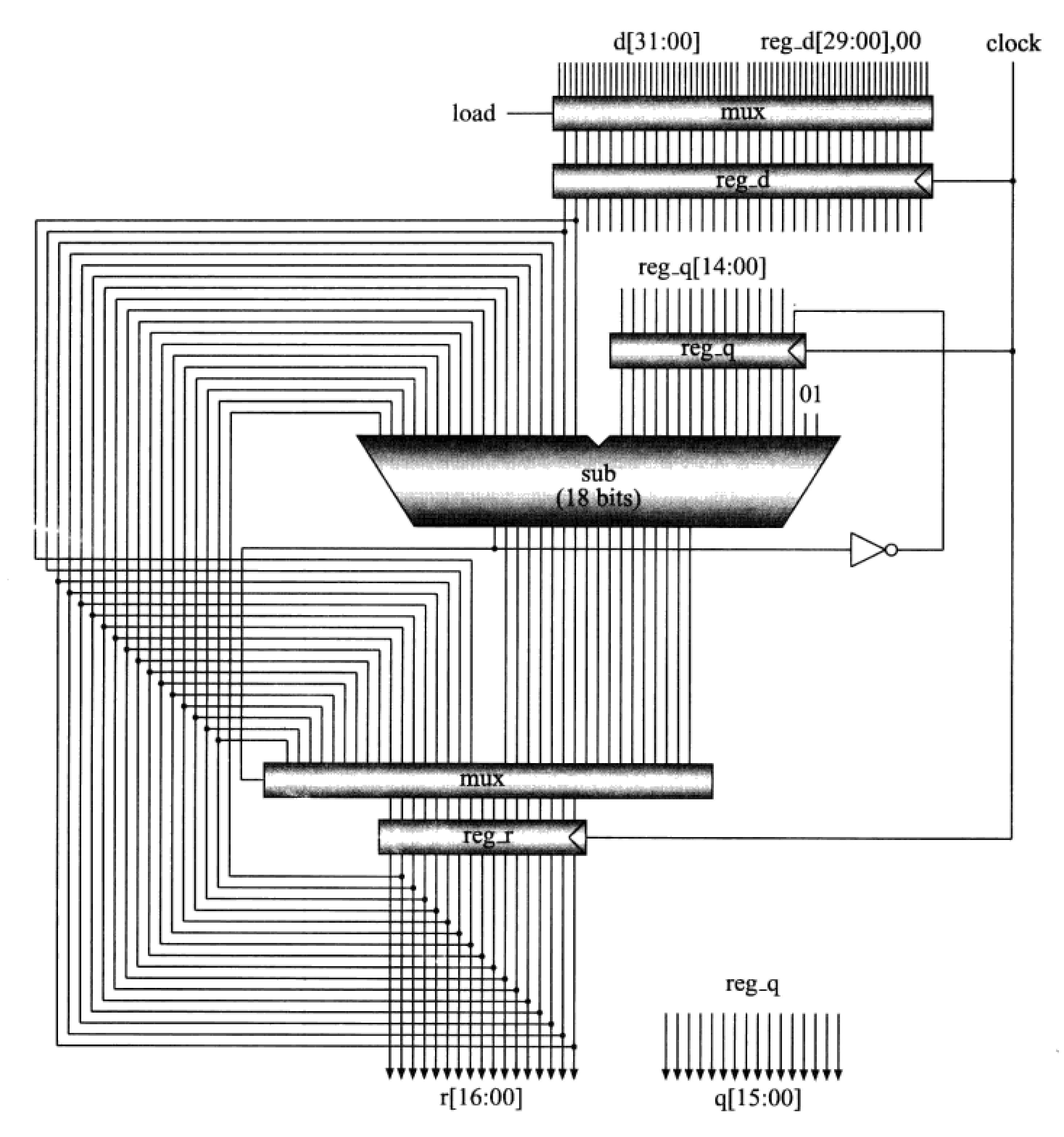


图 3.29 恢复余数开方电路总体图

器 reg_r 的输入端也需要一个多路器，选择相减结果或被减数。恢复余数开方电路的 Verilog HDL 代码如下。

```
module root_restoring (d,load,clock,resetn,q,r,busy,ready,count);
    input [31:00] d;     // radicand
    input load;          // ID stage: load = is_sqrt & ~busy;
    input clock,resetn;
    output [15:00] q;    // root
    output [16:00] r;    // remainder
    output busy;         // cannot receive new sqrt
    output ready;        // ready to save result
    output [4:0] count; // for sim test only
    reg [31:00] reg_d;
    reg [15:00] reg_q;
```

```
    reg [16:00] reg_r;
    reg [3:0] count;
    reg busy,busy2;
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count <= 4'b0;        // reset count
            busy  <= 0;           // reset to not busy
            busy2 <= 0;           // for generating 1-cycle ready
        end else begin            // not reset
            busy2 <= busy;        // 1-cycle delay of busy
            if (load) begin       // load: 1 cycle only
                reg_d <= d;       // load d
                reg_q <= 16'h0; // reset root
                reg_r <= 16'h0; // reset remainder
                count <= 4'b0;  // reset count
                busy  <= 1'b1;  // set to busy
             end else if (busy) begin // execution: 16 cycles
                reg_d <= {reg_d[29:00],2'b0}; // shift 2-bit
                reg_q <= {reg_q[14:00],~sub_out[17]}; // 1-bit q
                reg_r <= mux_out; // partial remainder
                count <= count + 4'b1;        // count++
                if (count == 4'hf) busy <= 0; // finish
            end
        end
    end
    assign ready = ~busy & busy2; // generate 1-cycle ready
    wire [17:00] sub_out = {reg_r[15:0],reg_d[31:30]}-{reg_q,2'b1};
    wire [16:00] mux_out = sub_out[17]?  // restoring or not
                          {reg_r[14:0],reg_d[31:30]}:sub_out[16:00];
    assign q = reg_q;
    assign r = reg_r;
endmodule
```

图 3.30 是恢复余数开方电路的仿真结果：$\mathrm{C0000000} = \mathrm{DDB3}^2 + \mathrm{174D7}$。

3.5.2 不恢复余数开方算法

在恢复余数开方算法中，如果部分余数为负，则要恢复原来的余数并左移两位。设第 i 次迭代时部分余数为 R_i，部分平方根为 Q_i。恢复余数相当于 $R_i + Q_i \times 4 + 1$，左移两位相当于 $(R_i + Q_i \times 4 + 1) \times 4$。由于部分余数为负，$Q_{i+1} = Q_i \times 2 + 0$。进入下一轮的迭代后，从部分余数中减去 $Q_{i+1} \times 4 + 1$。我们有以下的等式：

$$
\begin{aligned}
& (R_i + Q_i \times 4 + 1) \times 4 - (Q_{i+1} \times 4 + 1) \\
= & (R_i + Q_i \times 4 + 1) \times 4 - (Q_i \times 8 + 1) \\
= & R_i \times 4 + (Q_i \times 8 + 3) \\
= & R_i \times 4 + (Q_{i+1} \times 4 + 3)
\end{aligned}
$$

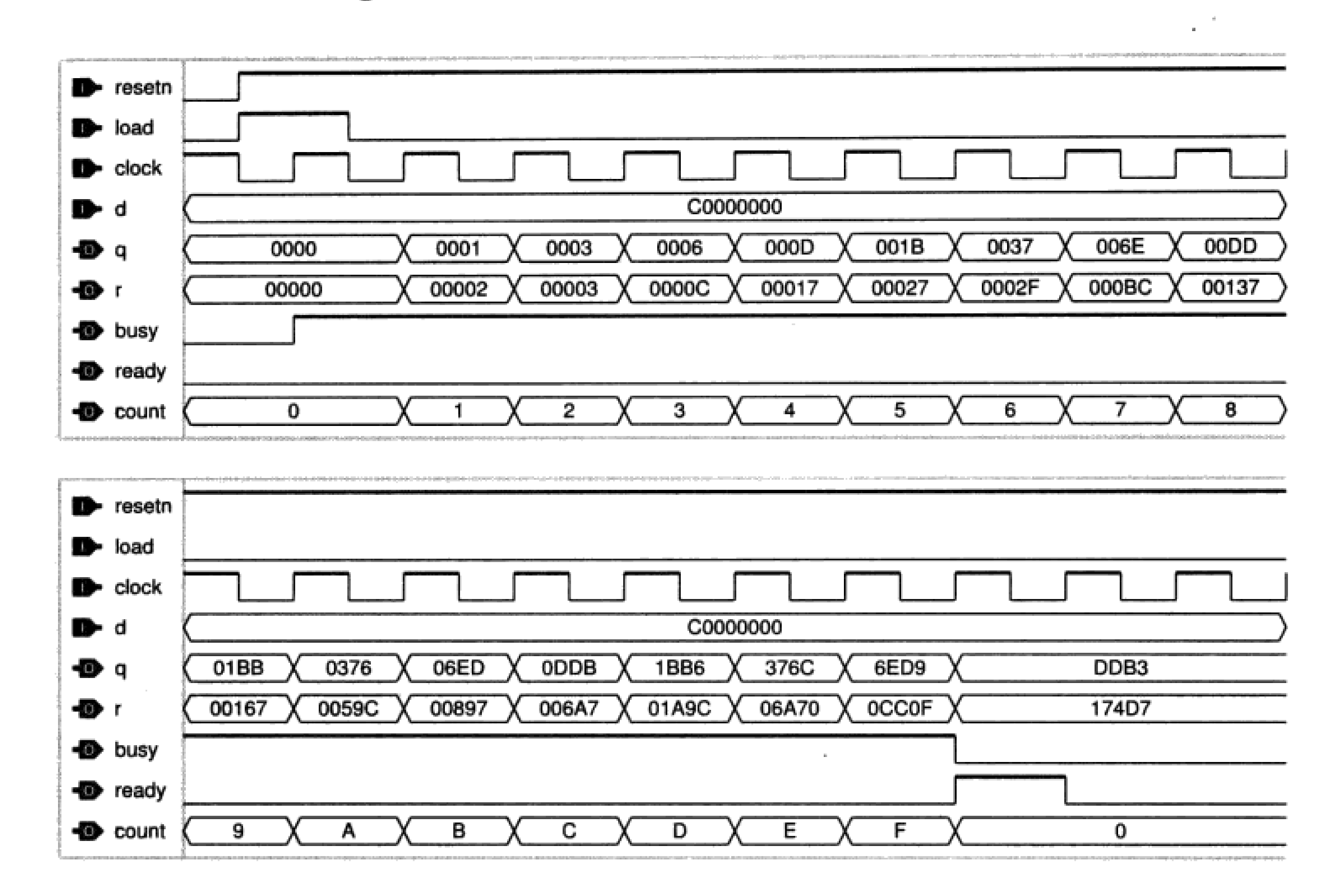


图 3.30 恢复余数开方电路仿真结果

这就是不恢复余数开方算法的中心思想[9, 12]。即，不管相减结果是正是负，都把它写入 reg_r，若为负，下次迭代不是从中减去 Q01 而是加上 Q11。以下的 C 程序实现不恢复余数开方算法。编译及运行结果紧跟着列出。

```
// Non-Restoring Square Root, Copyright by Li Yamin, yamin@ieee.org
#include <stdio.h>
unsigned squart(unsigned d, int *remainder) {
    unsigned q = 0;     // q: 16-bit unsigned integer (root)
    int r = 0;          // r: 17-bit integer (remainder)
    int i;
    for (i = 15; i >= 0; i-- ) {
        printf("%02d: q=%04x ",i,q);
        if (r >= 0) {
            r = ((r << 2) | ((d >> (i+i)) & 3)) - ((q << 2) | 1); // -q01
            printf("r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=%08x ",r);
        } else {
            r = ((r << 2) | ((d >> (i+i)) & 3)) + ((q << 2) | 3); // +q11
            printf("r=((r<<2)|((d>>(i+i))&3))+((q<<2)|3)=%08x ",r);
        }
        if (r >= 0) {q = (q << 1) | 1; printf("q=q*2+1=%04x\n",q);}
        else        {q = (q << 1) | 0; printf("q=q*2+0=%04x\n",q);}
    }
    if (r < 0) {r = r + ((q << 1) | 1); } // remainder adjusting
    *remainder = r; // return remainder
```

```
    return(q);      // return root
}

int main(void) {
    unsigned radicand,root;
    int remainder;
    printf("Input an unsigned hexedecimal number d = ");
    scanf("%x", &radicand);
    root = squart(radicand, & remainder);
    printf("hex: q = %08x, r = %08x, d = q*q + r = %08x\n",
            root,remainder,root*root + remainder);
    printf("dec: q = %08d, r = %08d, d = q*q + r = %08u\n",
            root,remainder,root*root + remainder);
}

[yamin@localhost cpu]$ gcc root_nonrestoring.c -o root_nonrestoring
[yamin@localhost cpu]$ root_nonrestoring
Input an unsigned hexedecimal number d  =  c0000000
15: q=0000 r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=00000002 q=q*2+1=0001
14: q=0001 r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=00000003 q=q*2+1=0003
13: q=0003 r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=ffffffff q=q*2+0=0006
12: q=0006 r=((r<<2)|((d>>(i+i))&3))+((q<<2)|3)=00000017 q=q*2+1=000d
11: q=000d r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=00000027 q=q*2+1=001b
10: q=001b r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=0000002f q=q*2+1=0037
09: q=0037 r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=ffffffdf q=q*2+0=006e
08: q=006e r=((r<<2)|((d>>(i+i))&3))+((q<<2)|3)=00000137 q=q*2+1=00dd
07: q=00dd r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=00000167 q=q*2+1=01bb
06: q=01bb r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=fffffeaf q=q*2+0=0376
05: q=0376 r=((r<<2)|((d>>(i+i))&3))+((q<<2)|3)=00000897 q=q*2+1=06ed
04: q=06ed r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=000006a7 q=q*2+1=0ddb
03: q=0ddb r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=ffffe32f q=q*2+0=1bb6
02: q=1bb6 r=((r<<2)|((d>>(i+i))&3))+((q<<2)|3)=fffffb97 q=q*2+0=376c
01: q=376c r=((r<<2)|((d>>(i+i))&3))+((q<<2)|3)=0000cc0f q=q*2+1=6ed9
00: q=6ed9 r=((r<<2)|((d>>(i+i))&3))-((q<<2)|1)=000174d7 q=q*2+1=ddb3
hex: q = 0000ddb3, r = 000174d7, d = q*q + r = c0000000
dec: q = 00056755, r = 00095447, d = q*q + r = 3221225472
```

图 3.31 是不恢复余数开方器总体电路图 (封面图的详细版)。图中左下角的加法器和多路器是为了得到正确的余数而设置的，如果在你的开方电路中不需要余数的话，可以扔掉它们。另外，如果做减法时 ci 是低电平有效，则可以将 −Q01 和 +Q11 统一成图中的 3 个逻辑门的电路 (加减法器少用了两位)。该电路使用的逻辑门很少，可用于低成本低功耗的嵌入式 CPU 的设计中。

以下是不恢复余数开方电路的 Verilog HDL 源代码。

```
module root_nonrestoring (d,load,clock,resetn,q,r,busy,ready,count);
    input   [31:0] d;     // radicand
```

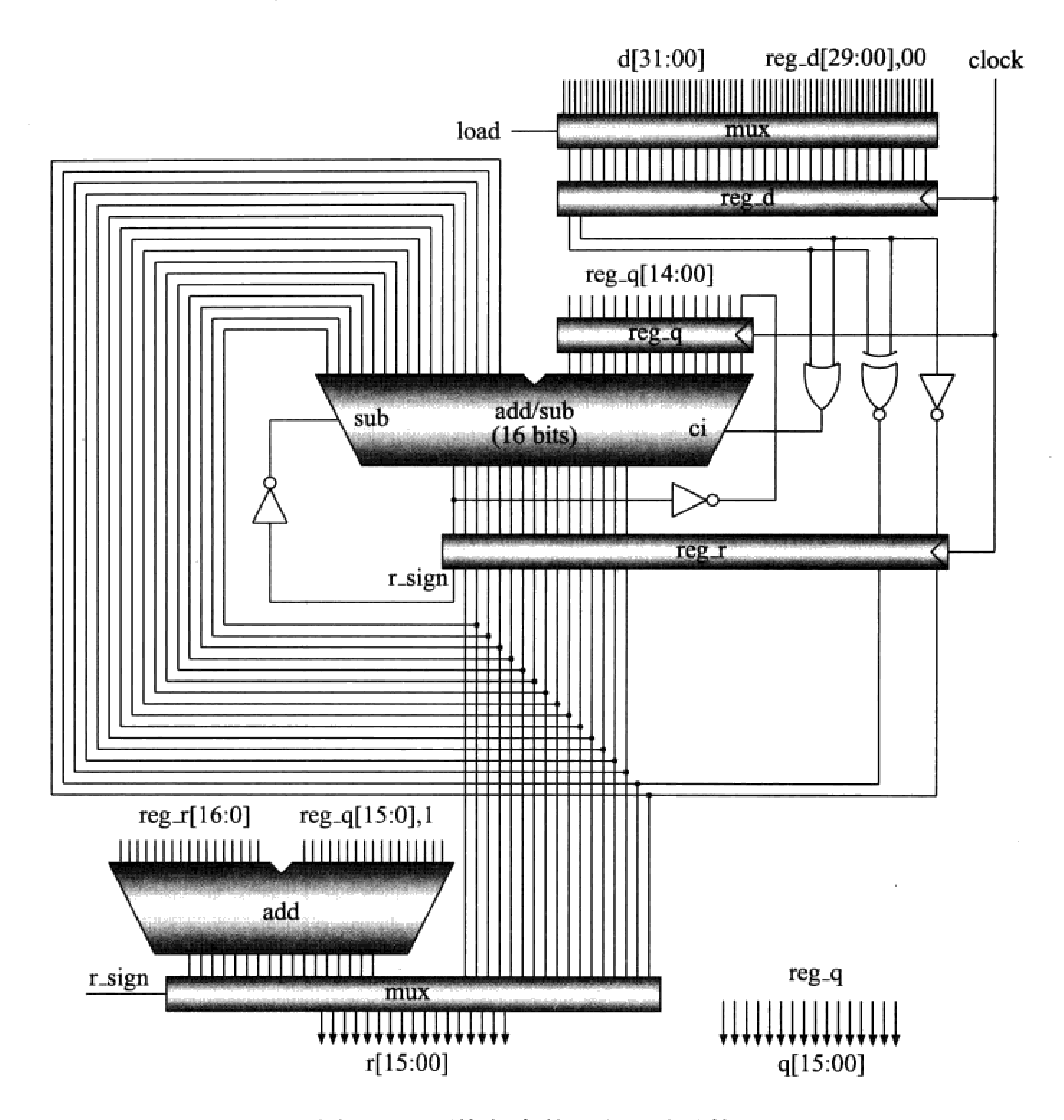


图 3.31 不恢复余数开方电路总体图

```
input  load;          // ID stage: load = is_sqrt & ~busy;
input  clock, resetn;
output [15:0] q;      // root
output [16:0] r;      // remainder
output busy;          // cannot receive new sqrt
output ready;         // ready to save result
output [4:0] count; // for sim test only
reg [31:0] reg_d;
reg [15:0] reg_q;
reg [17:0] reg_r;
reg  [3:0] count;
reg busy, busy2, r_sign;
always @ (posedge clock or negedge resetn) begin
    if (resetn == 0) begin
```

```
            count  <= 4'b0;       // reset count
            busy   <= 0;          // reset to not busy
            busy2  <= 0;          // for generating 1-cycle ready
        end else begin            // not reset
            busy2 <= busy;        // 1-cycle delay of busy
            if (load) begin       // load: 1 cycle only
                reg_d <= d;       // load d
                reg_q <= 16'h0;   // reset root
                reg_r <= 16'h0;   // reset remainder
                count <= 4'b0;    // reset count
                busy  <= 1'b1;    // set to busy
             end else if (busy) begin // execution: 16 cycles
                reg_d <= {reg_d[29:0],2'b0}; // shift 2-bit
                reg_q <= {reg_q[14:0],~add_sub[15]}; // 1-bit q
                reg_r[17:2] <= add_sub; // partial remainder
                reg_r[1] <= reg_d[31] ~^ reg_d[30]; // remainder
                reg_r[0] <= ~reg_d[30];             // remainder
                count    <= count + 4'b1;           // count++
                if (count == 4'hf) busy <= 0;       // finish
            end
        end
    end
    assign ready = ~busy & busy2; // generate 1-cycle ready
    wire [15:0] add_sub;
    // clas16 (sub,a,b,ci,s);      // active low for subtraction
    clas16 as (~reg_r[17],reg_r[15:0],reg_q,reg_d[31]|reg_d[30],
               add_sub);
    assign q = reg_q;
    assign r = reg_r[17] ?
               reg_r[16:0] + {reg_q[15:0],1'b1} :
               reg_r[16:0];  // adjust remainder
endmodule
```

以下是开方电路使用的先行进位加减法器。做加法时进位高电平有效，做减法时进位低电平有效。其中使用的模块 cla_16 已在本章开始处给出。

```
module clas16 (sub,a,b,ci,s);
    input sub; // 1- 0+
    input [15:0] a,b;
    input ci;  // active low for subtraction
    output [15:0] s;
    wire g_out,p_out;
    cla_16 cla (a,b^{16{sub}},ci,g_out,p_out,s);
endmodule
```

图 3.32 是不恢复余数开方电路的仿真结果。试与 C 程序运行结果进行比较。

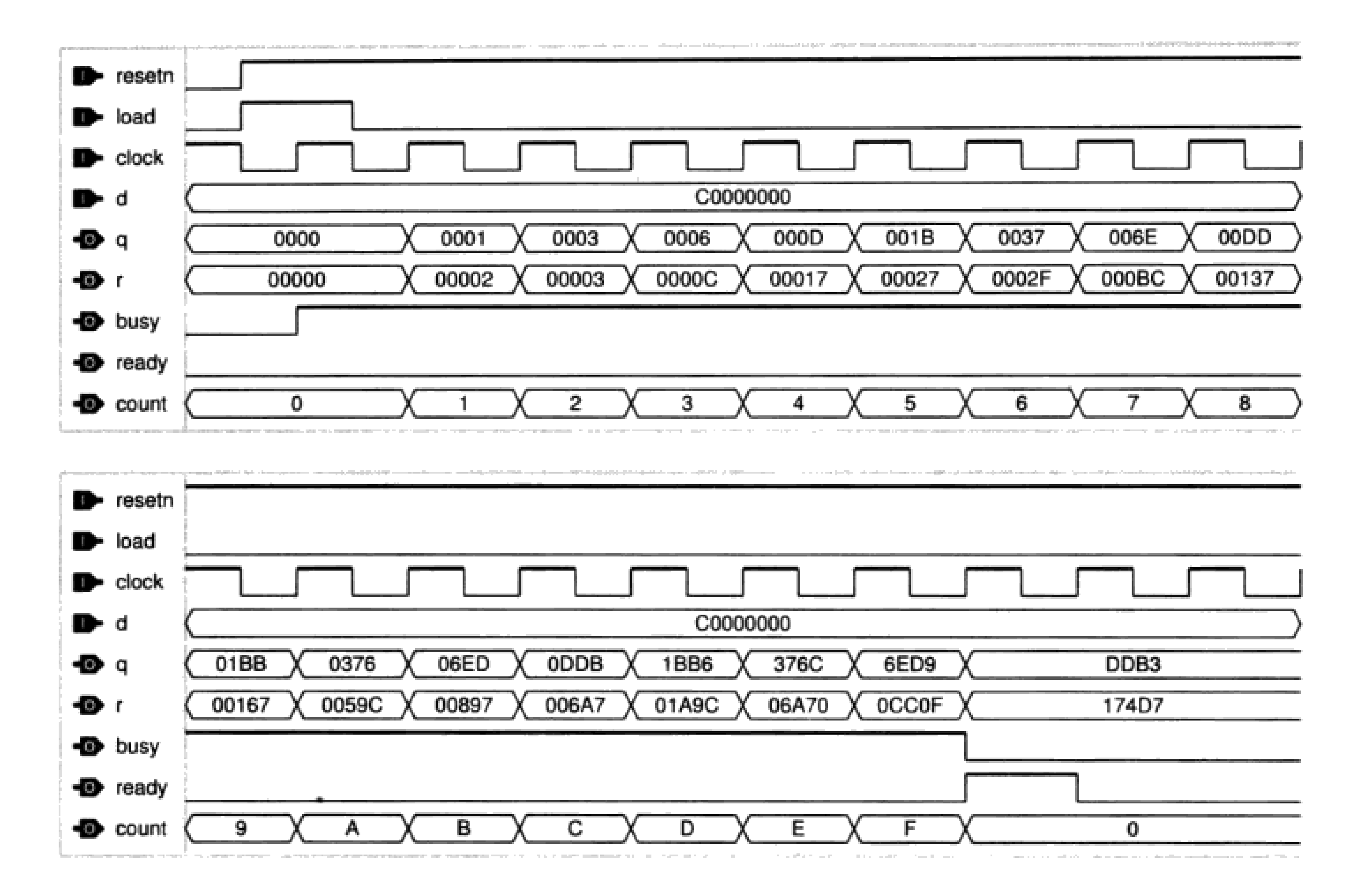


图 3.32 不恢复余数开方电路仿真结果

3.5.3 Goldschmidt 开方算法

设 $1/4 \leqslant d < 1$，即 d = .1xxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx 或 d = .01xx xxxx xxxx xxxx xxxx xxxx xxxx xxxx。以下介绍如何使用 Goldschmidt 算法计算 $\sqrt{d}$。如果分式

$$\frac{d \times r_0 \times r_1 \times r_2 \times \cdots \times r_{n-1}}{d \times r_0^2 \times r_1^2 \times r_2^2 \times \cdots \times r_{n-1}^2}$$

的分母趋向于 1，即 $r_0^2 \times r_1^2 \times r_2^2 \times \cdots \times r_n^2$ 趋向于 $1/d$、$r_0 \times r_1 \times r_2 \times \cdots \times r_n$ 趋向于 $\sqrt{1/d}$，则分子趋向于 $\sqrt{d}$。定义 $x_0 = d_0 = d$。通过查 ROM 表得到近似的 $r_0 \approx 1/\sqrt{d}$，或者直接令 $r_0 = 1 + (1 - d)/2$。然后使用下式进行迭代，直到分式的分母接近于 1 为止，这时的 $d_n \approx \sqrt{d}$。

$$\begin{aligned} r_i &= 1 + (1 - x_i)/2 \\ d_{i+1} &= d_i \times r_i \\ x_{i+1} &= x_i \times r_i^2 \end{aligned}$$

以下我们说明为什么 $x_n \to 1$。由于 $1/4 \leqslant d < 1$，我们有 $d = 1 - \delta$，$0 < \delta \leqslant 3/4$。令 $r_0 = 1 + \delta/2 = 1 + (1 - d)/2$，则 $x_1 = x_0 \times r_0^2 = (1 - \delta) \times (1 + \delta/2)^2 = (1 - \delta) \times (1 + \delta + \delta^2/4) \approx (1 - \delta) \times (1 + \delta) = 1 - \delta^2$。

一般地，$r_i = 1 + (1 - x_i)/2 \approx 1 + \delta^{2^i}/2$，则 $x_{i+1} = x_i \times r_i^2 = (1 - \delta^{2^i}) \times (1 + \delta^{2^i}/2)^2 = (1 - \delta^{2^i}) \times (1 + \delta^{2^i} + \delta^{2^{i+1}}/4) \approx (1 - \delta^{2^i}) \times (1 + \delta^{2^i}) = 1 - \delta^{2^{i+1}}$ 因为 $0 < \delta \leqslant 3/4$，所以 $x_{i+1} = 1 - \delta^{2^{i+1}} \to 1$，并且每迭代一次精度增加一倍。

Goldschmidt 开方电路的总体图如图 3.33 所示。我们使用了 ROM 来得到 r_0。三个寄存器 reg_r、reg_d 和 reg_x 分别存放 r_i、d_i 和 x_i。

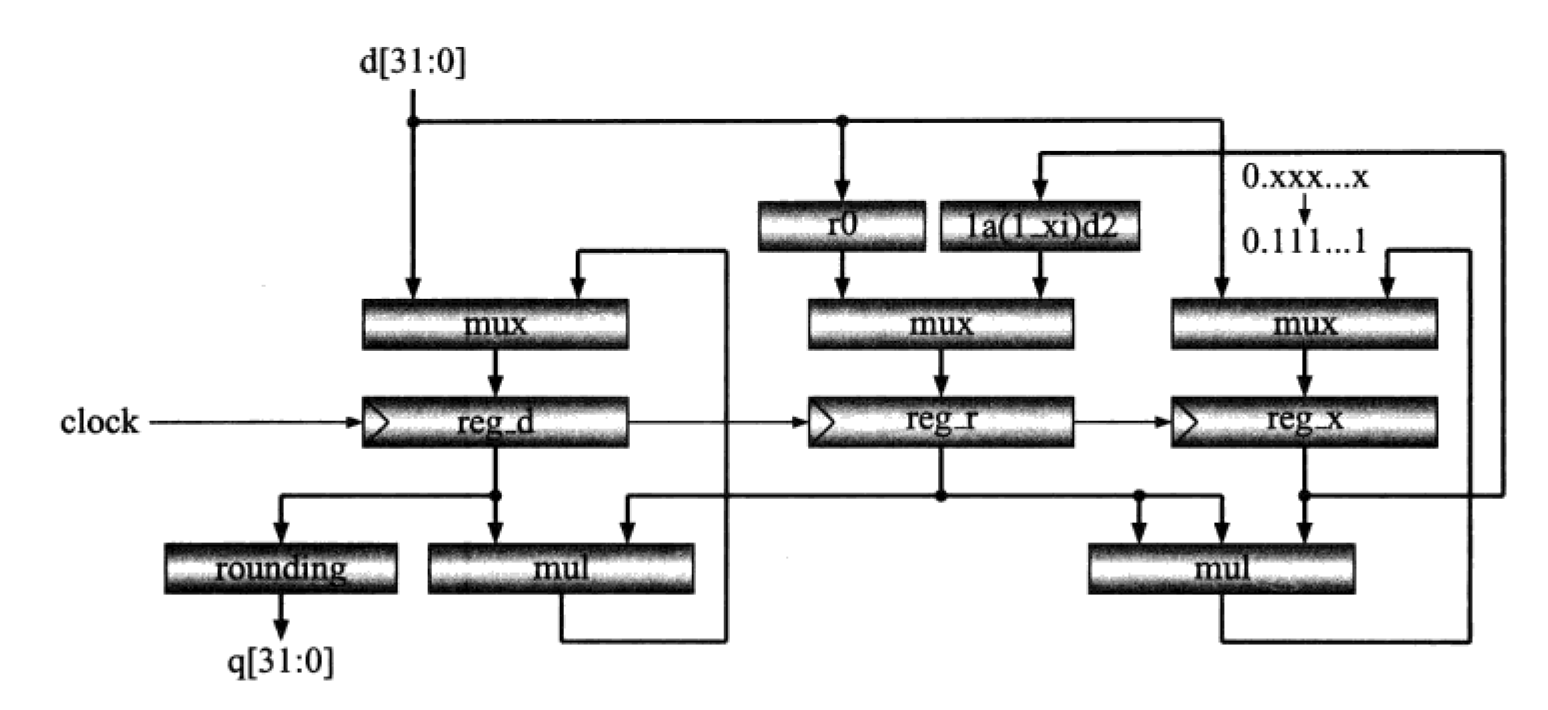


图 3.33　Goldschmidt 开方电路总体图

以下是实现 Goldschmidt 开方算法的 Verilog HDL 代码。由于我们在说明为什么 $x_n \to 1$ 时忽略了高阶项 $+\delta^{2^{i+1}}/4$，而在实际的运算中没有忽略，可能引起 $x_i \geqslant 1$，因此我们在代码中调整了 x_i，使其永远小于 1。

```
module root_goldschmidt (d,start,clock,resetn,q,busy,ready,count,
                         reg_x,);
    input [31:00] d;      // radicand: .1xxx...x or .01xx...x
    input start;          // ID stage: start = is_sqrt & ~busy;
    input clock,resetn;
    output [31:00] q;     // root: .1xxx...x
    output busy;          // cannot receive new sqrt
    output ready;         // ready to save result
    output [2:0] count;   // for sim test only
    output [34:00] reg_x; // for sim test only
    reg [34:00] reg_d;    // 35-bit: x.xxxx...xx
    reg [34:00] reg_x;    // 35-bit: 0.1xxx...xx
    reg [34:00] reg_r;    // 35-bit: x.xxxx...xx
    reg [2:0] count;      // 5 iterations
    reg busy,busy2;
    wire [7:0] r0 = rom(d[31:28]);
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count <= 3'b0;    // reset count
            busy  <= 0;       // reset to not busy
            busy2 <= 0;       // for generating 1-cycle ready
        end else begin        // not reset
```

```
            busy2 <= busy;    // 1-cycle delay of busy
            if (start) begin // start: 1 cycle only
                reg_d <= {1'b0, d, 2'b0};
                reg_x <= {1'b0, d, 2'b0};
                reg_r <= {1'b1,r0,26'h0}; // 1.xxxx...x0
                count <= 3'b0;            // reset count
                busy  <= 1'b1;            // set to busy
             end else begin  // execution: 5 iterations
                reg_d <= d70[68:34];      // x.xxx...x
                reg_x <= x35;             // x.xxx...x
                reg_r <= ri;              // x.xxx...x
                count <= count + 3'b1;    // count++
                if (count == 3'h5) busy <= 0; // finish
            end
        end
    end
    wire [34:00] ri  = 35'h600000000-{2'b0,reg_x[33:1]};
    wire [69:00] ci = ri * ri;            // 0x.xxx...xx
    wire [69:00] d70 = reg_d * ri;        // 01.xxx...xx
    wire [69:00] x70 = reg_x * ci[68:34]; // 0x.xxx...xx
    wire [34:00] x35 = {1'b0,{34{x70[68]}}|x70[67:34]}; // !>=1.0
    assign ready = ~busy & busy2; // generate 1-cycle ready
    assign q = reg_d[33:02] + {31'h0,(reg_d[01]|reg_d[00])};
    function [7:0] rom; // 1/d^{1/2}
        input [3:0] d;
        case (d)
            4'h0: rom = 8'hf0;              4'h1: rom = 8'hd4;
            4'h2: rom = 8'hba;              4'h3: rom = 8'ha4;
            4'h4: rom = 8'h8f;              4'h5: rom = 8'h7d;
            4'h6: rom = 8'h6c;              4'h7: rom = 8'h5c;
            4'h8: rom = 8'h4e;              4'h9: rom = 8'h41;
            4'ha: rom = 8'h35;              4'hb: rom = 8'h29;
            4'hc: rom = 8'h1f;              4'hd: rom = 8'h15;
            4'he: rom = 8'h0c;              4'hf: rom = 8'h04;
        endcase
    endfunction
endmodule
```

图 3.34 给出 Goldschmidt 开方电路的仿真结果。第一个例子示出的是 $d = 0.25$，$q = 0.5$。0.25 是 d 所允许的最小值。由于它离 1.0 最远，所需的迭代次数也最多 (6 次)。第二个例子是 $d = 0.75$ 的情况，读者可以将它的输出与图 3.32 所示的整数开方结果进行比较。第二个例子只需 4 次迭代结果就出来了。如果要固定电路的迭代次数，当然要考虑最坏的情况，因此我们规定 Goldschmidt 开方电路需要 6 次迭代。

每次迭代最费时间的是 $x_i \times r_i \times r_i$ (两次乘法)。即使我们采用 Wallace 树型乘法器，也至少需要 4 个加法器周期，加上开始的查表 1 个周期及最后舍入的 1 个周期，

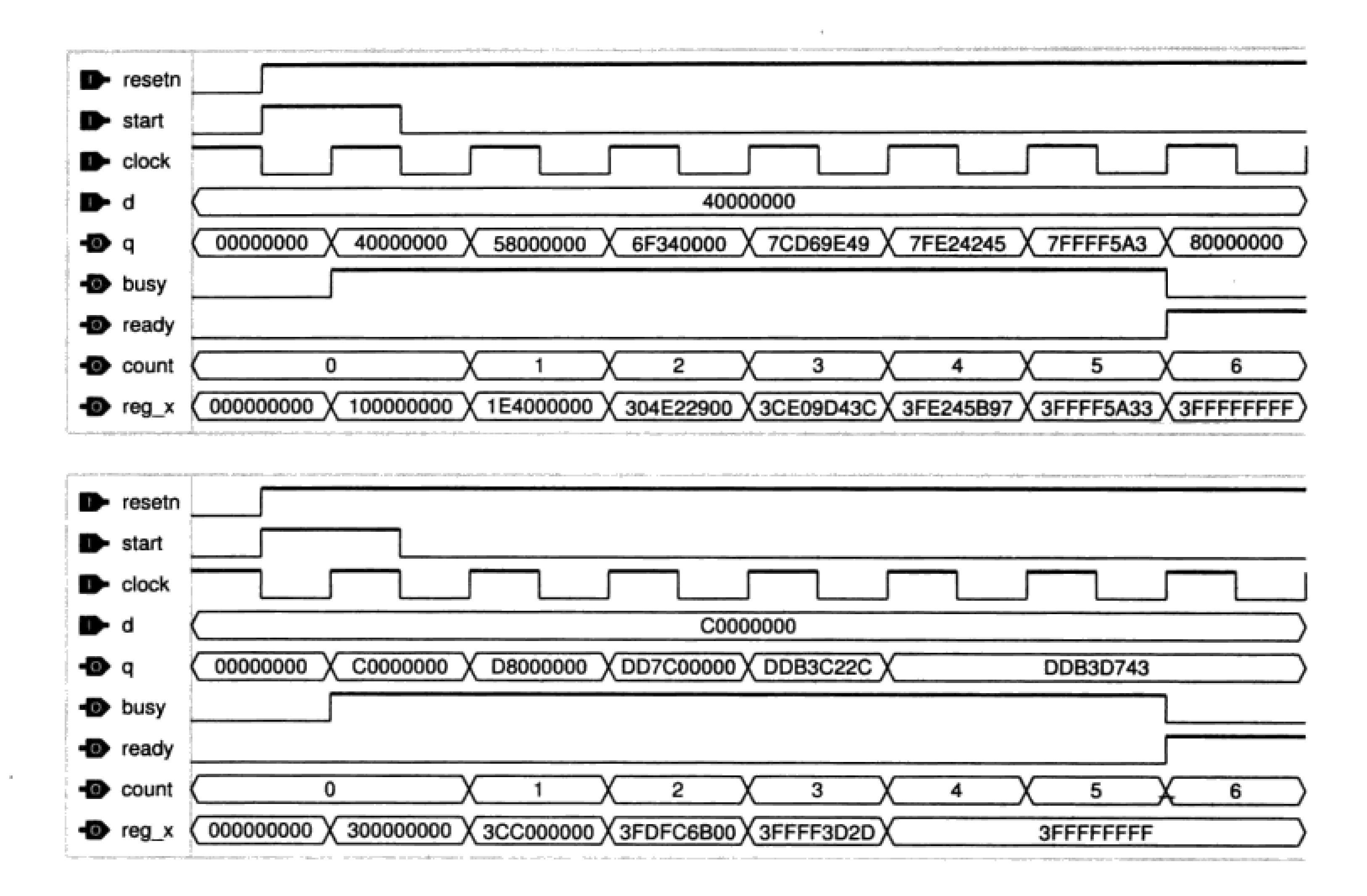


图 3.34 Goldschmidt 开方电路仿真结果

总共需要 26 个周期。作为参考，IBM G4 服务器的浮点部件采用 Goldschmidt 开方算法，对 35 位数据开方时也是需要 26 个周期[29]。

3.5.4 Newton-Raphson 开方算法

设 $1/4 \leqslant d < 1$，即，$d = 0.1xx\cdots x$ 或 $d = 0.01x\cdots x$，计算 $q = \sqrt{d}$。参照 Newton-Raphson 除法算法，令 $f(x) = 1/x^2 - d$，则在 $x = 1/\sqrt{d}$ 处 $f(x) = 0$。应用牛顿迭代公式，我们有:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i(3 - x_i^2 d)/2$$

注意 x_i 是 $1/\sqrt{d}$ 的近似值。经过 n 次迭代，得到有足够精度的 $1/\sqrt{d}$ 的近似值 x_n 后，$\sqrt{d}$ 可由 $\sqrt{d} \approx d \times x_n$ 算出。

图 3.35 是 Newton-Raphson 开方电路总体图，寄存器 reg_d 和寄存器 reg_x 分别存放 d 和 x_i。Verilog HDL 代码如下。

```
module root_newton (d,start,clock,resetn,q,busy,ready,count,reg_x);
    input [31:00] d;       // radicand: .1xxx...x or .01xx...x
    input start;           // ID stage: start = is_sqrt & ~busy;
    input clock,resetn;
    output [31:00] q;      // root: .1xxx...x
```

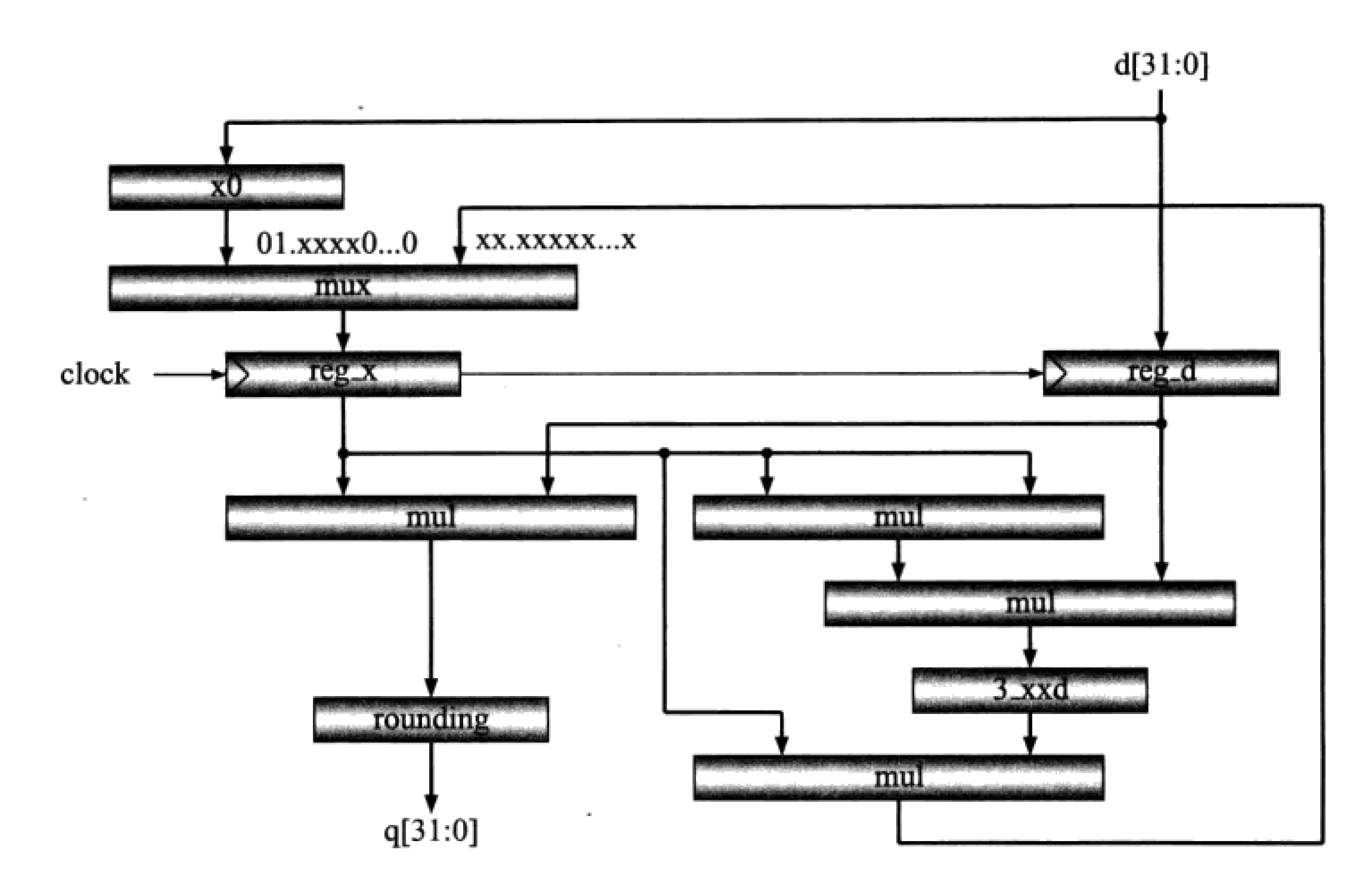


图 3.35 Newton-Raphson 开方电路总体图

```
output busy;           // cannot receive new sqrt
output ready;          // ready to save result
output [1:0] count;    // for sim test only
output [33:00] reg_x;  // for sim test only
reg [31:00] reg_d;     // 32-bit:   .xxxx...xx
reg [33:00] reg_x;     // 34-bit: xx.1xxx...xx
reg [1:0] count;       // 2 iterations
reg busy,busy2;
wire [7:0] x0 = rom(d[31:27]);
always @ (posedge clock or negedge resetn) begin
    if (resetn == 0) begin
        count <= 2'b0;    // reset count
        busy  <= 0;       // reset to not busy
        busy2 <= 0;       // for generating 1-cycle ready
    end else begin        // not reset
        busy2 <= busy;    // 1-cycle delay of busy
        if (start) begin // start: 1 cycle only
            reg_x <= {2'b1,x0,24'b0};  // 01.xxxx0...0
            reg_d <= d;                //    .1xxxx...x
            count <= 2'b0;             // reset count
            busy  <= 1'b1;             // set to busy
         end else begin  // execution: 3 iterations
            reg_x <= x68[66:33];       // x68/2
            count <= count + 2'b1;     // count++
            if (count == 2'h2) busy <= 0; // finish
```

```
            end
        end
    end
    // x_{i+1} = x_i * (3 - x_i * x_i * d) / 2
    wire [67:00] x_2 = reg_x * reg_x;                 // xxxx.xxxx...x
    wire [67:00] x2d = reg_d * x_2[67:32];            // xxxx.xxxx...x
    wire [33:00] b34 = 34'h300000000 - x2d[65:32];//    xx.xxxx...x
    wire [67:00] x68 = reg_x * b34;                   // xxxx.xxxx...x
    // q = d * x_n
    assign ready = ~busy & busy2; // generate 1-cycle ready
    wire [65:00] d_x = reg_d * reg_x;                 //    xx.xxxx...x
    assign q = d_x[63:32] + {31'h0,|d_x[31:0]};       // rounding
    function [7:0] rom; // 1/d^{1/2}
        input [4:0] d;
        case (d)
            5'h08: rom = 8'hf0;           5'h09: rom = 8'hd5;
            5'h0a: rom = 8'hbe;           5'h0b: rom = 8'hab;
            5'h0c: rom = 8'h99;           5'h0d: rom = 8'h8a;
            5'h0e: rom = 8'h7c;           5'h0f: rom = 8'h6f;
            5'h10: rom = 8'h64;           5'h11: rom = 8'h5a;
            5'h12: rom = 8'h50;           5'h13: rom = 8'h47;
            5'h14: rom = 8'h3f;           5'h15: rom = 8'h38;
            5'h16: rom = 8'h31;           5'h17: rom = 8'h2a;
            5'h18: rom = 8'h24;           5'h19: rom = 8'h1e;
            5'h1a: rom = 8'h19;           5'h1b: rom = 8'h14;
            5'h1c: rom = 8'h0f;           5'h1d: rom = 8'h0a;
            5'h1e: rom = 8'h06;           5'h1f: rom = 8'h02;
            default: rom = 8'hff;
        endcase
    endfunction
endmodule
```

我们为 x_0 准备了 8 位初值但只使用了 5 位地址。实际上为了得到 x_0 的 8 位准确初值，5 位输入地址是不够的，有兴趣的读者可以探讨一下到底需要几位地址才行。Newton-Raphson 开方电路仿真结果如 3.36 图所示。

每次迭代所需的周期为：3 次乘法需 6 个周期，一次减法需 1 个周期。3 次迭代共需 21 个周期。最后还有一次乘法及一次舍入，以及开始处的查表，总共需 25 个周期。实际上，大多数的 CPU 也均是采用 3 次迭代。

表 3.5 比较了 Goldschmidt 和 Newton-Raphson 算法实现除法和开方操作时需要的时钟周期数量。如果将这些算法用于浮点数操作，最后的一次舍入可以不做，因为反正在浮点部件中最后总是有一次舍入操作 (规格化)。另外，Goldschmidt 开方算法中有三个数相乘且其中的两个数相同，我们可以想办法用三个加法周期完成。这样的话，Goldschmidt 开方算法只需 $1 + 6 \times 3 + 1 = 20$ 个时钟周期。我们将在浮点部件 FPU 的设计中使用 Newton-Raphson 算法完成浮点除法和浮点开方运算。

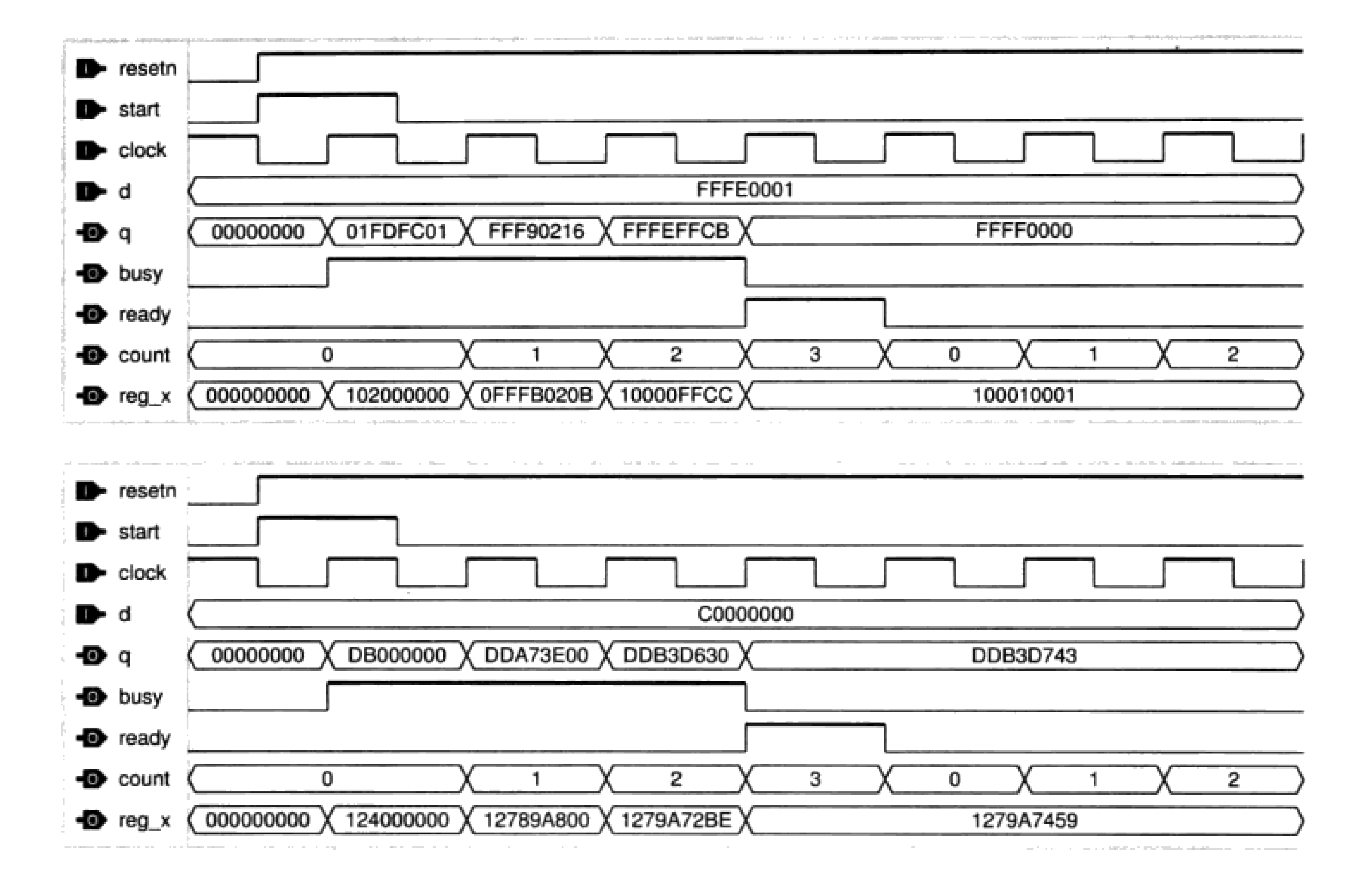


图 3.36 Newton-Raphson 开方电路仿真结果

表 3.5 Goldschmidt 和 Newton-Raphson 算法所需时钟周期数量

算 法	除 法	开 方
Goldschmidt	$5 \times 3 + 1 = 16$	$1 + 6 \times 4 + 1 = 26$
Newton-Raphson	$1 + 3 \times 5 + 2 + 1 = 19$	$1 + 3 \times 7 + 2 + 1 = 25$

3.6 习题

1. 试设计一个 32 位的加减法器，增加一个输出信号 v 来指出结果是否上溢。
2. 除了本章描述的先行进位加法器，调查还有哪些其他结构的先行进位加法器。
3. 用 Verilog HDL 实现迭代方法的乘法操作。
4. 我们已经讨论了 $a \times b$ 的各种算法。若 $b == a$，则 $a \times b = a^2$。试考虑平方运算的特殊性以及如何利用这些特殊性加快运算速度并减少所需全加器的数量。
5. 试考查 Booth 乘法算法并用 Verilog HDL 加以实现。
6. 试证明 Goldschmidt 和 Newton-Raphson 除法算法每迭代一次精度增加一倍。
7. 试用 C 语言写一个程序，用来计算 Goldschmidt 或 Newton-Raphson 除法电路中 ROM 应存放的数据。
8. 出道数学题：证明十进制手算开方算法的正确性。
9. 试证明图 3.31 电路中使用右边 3 个逻辑门的正确性。

10. 图 3.33 中使用了三个寄存器。试设计一个电路，不使用 ROM，且只使用两个寄存器来实现 Goldschmidt 开方算法。
11. 考查什么是 SRT 除法和开方算法，并使用 Verilog HDL 实现 SRT 除法或开方算法 (要求：Radix $\geqslant 4$)。
12. 试考查 CORDIC (Coordinate Rotation Digital Computer) 算法以及如何用电路实现初等超越函数 (例如三角函数、指数函数、对数函数等) 的运算。

第 4 章　指令系统及 ALU 设计

一个 CPU 的指令系统 (Instruction Set) 定义该 CPU“所能做的”或者“应该做的”工作。“所能做的”意思是告诉系统软件 (比如编译器) 设计者该 CPU 能做什么事情 (软件设计者使用它)；“应该做的”意思是告诉硬件设计者在设计 CPU 时应该让 CPU 完成的任务 (硬件设计者实现它)。也就是说，指令系统是软件设计者和硬件设计者之间能对上话的一个“接口”，如图 4.1 所示。

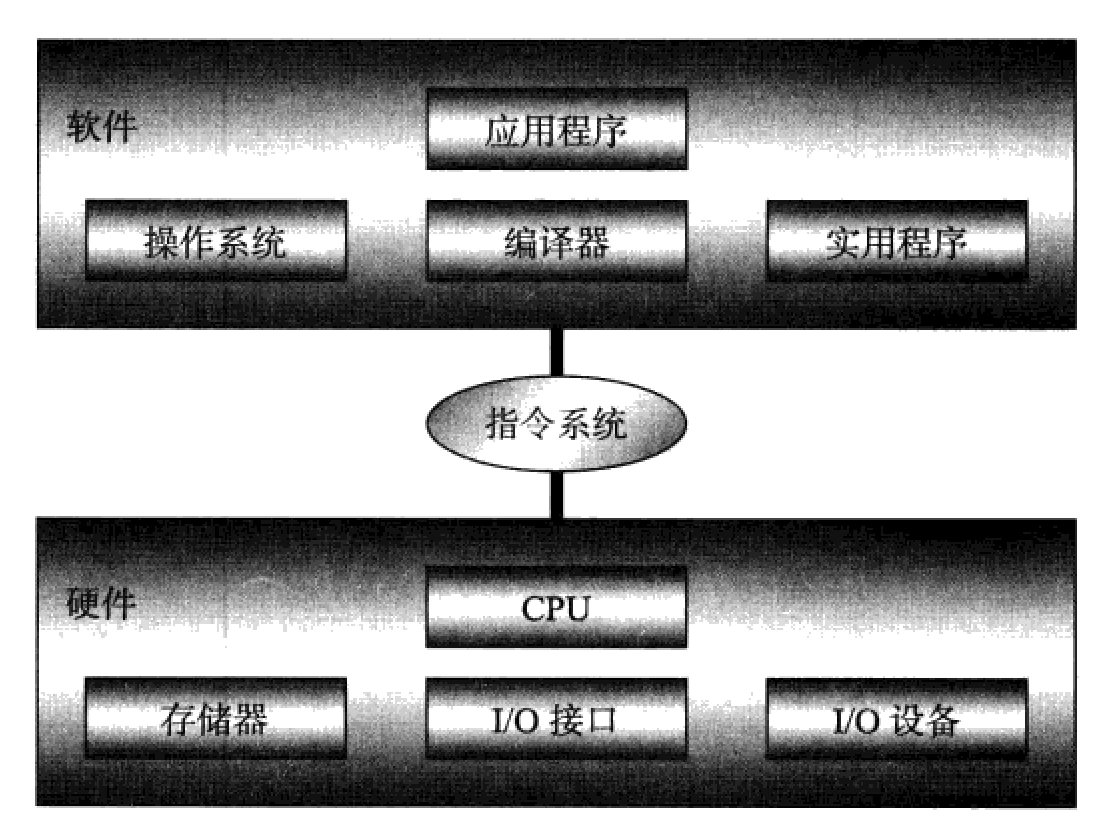


图 4.1　指令系统是硬件与软件之间的接口

4.1　指令系统结构

CPU 不能直接执行用高级语言编写的程序。我们需要一个编译器 (Compiler) 把这个程序转换成计算机可执行的二进制代码。二进制代码主要包括指令和数据。指令系统结构 ISA (Instruction Set Architecture) 定义指令的格式、指令的意义、操作数的类型和指令能访问的寄存器与存储器。

目前常用的指令系统结构有 Intel 的 x86、SGI/MIPS 的 MIPS32/MIPS64、IBM 的 PowerPC、SUN Microsystems 的 SPARC、DEC 的 Alpha、HP 的 HP-PA 等。除了 x86，上述指令系统结构均属于 RISC 类型。

4.1.1　操作数类型

指令的主要任务是对操作数进行计算。操作数有不同的类型及相应的长度 (二进制位数)。表 4.1 列出了常用的数据类型。表中也列出了 C 语言使用这些数据类型时的关键字。我们将在第 9 章介绍浮点数。

表 4.1 常用的数据类型

数据类型	位数	数值范围	C 语言中的对应
字节	8	−128 ～ +127	signed char
无符号字节	8	0 ～ 255	unsigned char
半字	16	−32 768 ～ +32 767	short int
无符号半字	16	0 ～ 65 535	unsigned short int
字	32	−2 147 483 648 ～ +2 147 483 647	int
无符号字	32	0 ～ 4 294 967 295	unsigned int
单精度浮点数	32		float
双精度浮点数	64		double

4.1.2 数据在存储器中的存放方法

存储器是以字节为单位编址的。一个字有 4 个字节，它在存储器中的存放方法有两种：Little Endian 和 Big Endian，如图 4.2 所示。0x 表示是十六进制。

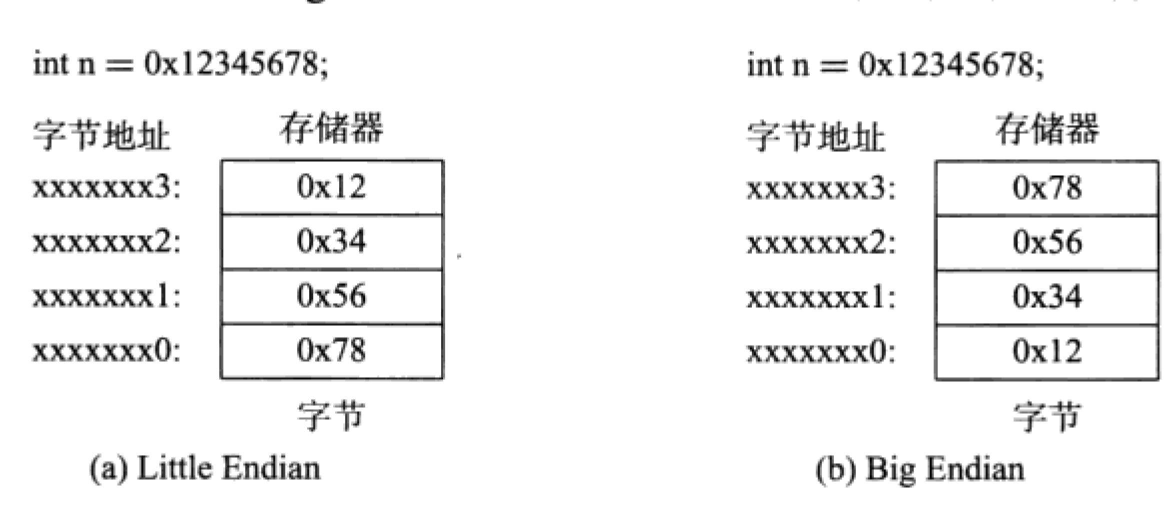


图 4.2 数据在存储器中的存放方法

图中的字 n = 0x12345678，它有 4 个字节，从高到低依次为 0x12，0x34，0x56 和 0x78。在 Little Endian 方式中，低字节放入低地址单元，高字节放入高地址单元。而 Big Endian 则相反，高字节放入低地址单元，低字节放入高地址单元。以下的 C 程序是利用地址指针来判断数据在存储器中是如何存放的。

```
main () {
    int n = 0x12345678;
    if (*(char *)&n == 0x78)
         printf("little endian\n");
    else printf("big endian\n");
}
```

如果读者不怎么熟悉指针类型，也可使用 union 来判断，见下面的 C 程序。一个 union 中的所有变量共有一个实体，即一个数据可用不同的方式来访问，就像一个人有好多名字一样。

```
main() {
    union {
        int intword;
        char characters[sizeof (int)];
    } u;
    u.intword = 0x12345678;
    if (u.characters[sizeof (int) - 1] == 0x12)
         printf("little endian\n");
    else printf("big endian\n");
}
```

不管是 Little Endian 还是 Big Endian，我们说图 4.2 中一个字在存储器中的存放是地址对准的。即，4 个字节有相同的字地址[1]。否则，我们说地址没对准。半字也有地址对准的问题，但字节没有这个问题，如图 4.3 所示。

字节地址的低 3 位

Big Endian:	0	1	2	3	4	5	6	7
Little Endian:	7	6	5	4	3	2	1	0
	字节	字节	字节	字节	字节	字节	字节	字节
	半字		半字		半字		半字	
	字 / 单精度浮点数				字 / 单精度浮点数			
	双精度浮点数							

图 4.3 地址对准

有些 ISA 允许没对准的数据存放。如果不允许但出现了数据没对准的情况，比如访问一个字时，地址的最低两位不是 0，则硬件应该产生异常信号来通知 CPU。

4.1.3 指令类型

CPU 执行指令时必须首先把指令从存储器中取来。CPU 中有一个程序计数器 PC (Program Counter)。取指令时使用 PC 作为存储器的地址。不同的 ISA 有不同的指令系统。一般而言，我们把所有的指令分为以下 8 种类型。注意，以下括号中出现的指令助记符采用了意义比较明显的一般的符号，而并不是针对哪一种特定的 CPU 的指令系统。

1. 算术运算类型

算术运算类型通常是指对整数的运算。基本的运算包括加 (add)、减 (sub)、乘 (mul)、除 (div) 等。

---

[1]字节地址去掉最低两位就是字地址。

2. 逻辑运算类型

基本的逻辑运算只有 3 种：与 (and)、或 (or) 和非 (not)，但大多数指令系统不提供 not 指令，而是用异或 (xor) 来代替 not。

3. 移位操作类型

移位操作分左移和右移两种。左移 (shl) 是把操作数向左移若干位，右边空出的位填 0。右移分为逻辑右移 (shr) 和算术右移 (shra)。逻辑右移在左边空出的位填 0。算术右移在左边空出的位填右移前的操作数的最高位，相当于符号扩展。除了以上三种移位指令，还有循环左右移位以及连同进位位 C 的 (循环) 左右移位等。

4. 存储器访问类型

存储器访问指令有取存储器数据 (load) 和存存储器数据 (store) 两种。load 指令根据计算出的存储器地址从存储器读出数据，然后把数据写入 CPU 内部的寄存器。store 指令则相反，把寄存器中的数据写入存储器。这两种指令一定会出现在 RISC 类型的 ISA 中，而 CISC 类型的 ISA 可能会把 load/store 操作合并在运算类型的指令中。load 和 store 指令可以搬运的数据类型有字节、半字、字 (单精度浮点数) 和双字 (双精度浮点数) 等。字节和半字还有扩展的问题。

5. I/O 访问类型

I/O 指令访问 I/O 端口，有读 I/O 端口 (input) 指令和写 I/O 端口 (output) 指令。与存储器访问指令不同，output 指令往一个端口写入的可能是控制信息，而 input 指令从相同的端口读出的可能是状态信息。I/O 访问指令只出现在具有分开的 I/O 空间和存储器空间的 ISA 中，而大多数 RISC 类型的 ISA 使用单一的地址空间，把其中的一部分空间分给 I/O，同样使用 load/store 指令来访问它们。我们称这种方式具有存储器映像的 I/O 空间。

6. 转移控制类型

转移控制类型的指令有以下几种。条件转移 (beq、bne 等) 指令判断条件是否成立。若成立，则转移到目标地址去执行。对 C 语言程序中的 if 语句进行编译时会用到这些指令。目标地址可以通过计算得出，比如 PC 加上一个带符号的偏移量。跳转 (jump) 指令无条件地转移到目标地址去执行。目标地址可以在指令中明确给出，也可以是寄存器中的内容。子程序调用 (call) 指令跳转到子程序去执行。跳转的同时要把返回地址保存在某个地方，以便从子程序返回时使用。返回 (return) 指令把 call 指令保存的返回地址写入 PC，实现从子程序的返回。

7. 浮点运算类型

浮点 (Floating Point) 运算类型指令对浮点数进行运算，例如：浮点加 (fadd)、浮点减 (fsub)、浮点乘 (fmul)、浮点除 (fdiv)、浮点开方 (fsqrt)、整数与单精度浮点数之

间的转换 (i2f 和 f2i)、整数与双精度浮点数之间的转换 (i2d 和 d2i)、单精度浮点数与双精度浮点数之间的转换 (f2d 和 d2f) 等。

8. 系统控制类型

系统控制类型指令主要包括对 CPU 的状态寄存器和控制寄存器的读写、操作系统的调用及异常或中断处理程序的调用和返回等。

4.1.4 指令结构

目前常用的指令结构有以下三种：

1) 堆栈 (Stack) 结构：两个操作数总是在堆栈的栈顶。
2) 累加器 (Accumulator) 结构：一个操作数总是在累加器中。
3) 通用寄存器 (General Purpose Register) 结构：
   (1) 寄存器 - 存储器结构：一个操作数在寄存器中，另一个在存储器中；
   (2) 寄存器 - 寄存器结构：两个操作数都在寄存器中。

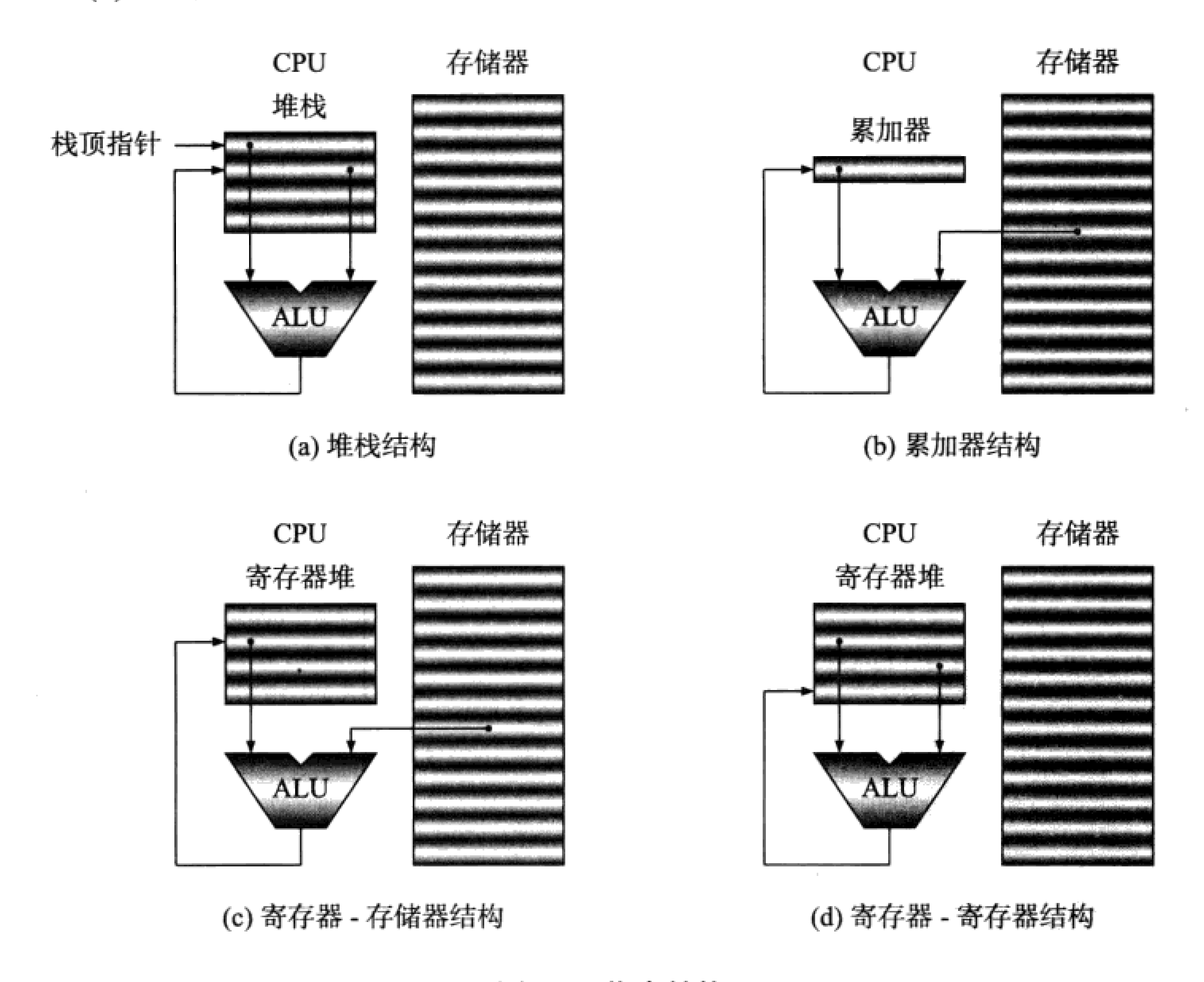


图 4.4 指令结构

图 4.4 是以上三种指令结构在执行计算类型指令时的示意图。图中的 ALU 是负责计算的电路。堆栈的操作主要有压入 (Push) 和弹出 (Pop) 两种。它有一个栈顶指

针，通常命名为 SP (Stack Pointer)，指出当前栈顶的位置。压入和弹出操作自动修改栈顶指针。堆栈具有后进先出 (Last-In First-Out) 或者先进后出 (First-In Last-Out) 的特点，就像往玻璃试管中放硬币似的[2]。

累加器是一个特殊的寄存器，对它的访问隐含在指令的操作码中。通用寄存器合在一起称为寄存器堆。现在的 CPU 的通用寄存器个数从十几个到上百个不等。对哪个寄存器进行访问，要在指令中明确指出来。

表 4.2 给出了这三种指令结构实现 Z = X + Y 的指令代码，其中 X、Y 和 Z 均在存储器中。

表 4.2 实现 Z = X + Y 的三种指令结构的指令代码

堆栈结构	累加器结构	通用寄存器结构	
		寄存器 - 存储器结构	寄存器 - 寄存器结构
push X	load X	load r1, X	load r1, X
push Y	**add Y**	**add r1, Y**	load r2, Y
add	store Z	store Z, r1	**add r3, r1, r2**
pop Z			store Z, r3

在堆栈结构的代码中，前两条指令把存储器操作数压入堆栈中；第三条指令弹出栈顶的两个操作数、相加、结果再压入堆栈；第四条指令弹出栈顶的数据，把它存入存储器。在累加器结构的代码中，第一条指令把存储器操作数取到累加器中；第二条指令把累加器中的操作数与存储器操作数相加，结果存入累加器；第三条指令把累加器的数据存入存储器。在寄存器 - 存储器结构的代码中，第一条指令把存储器操作数取到 r1 寄存器中 (当然也可以使用别的寄存器，寄存器号要在指令中明确给出)；第二条指令把 r1 寄存器中的操作数与存储器操作数相加，结果存入 r1 寄存器 (如果是 3 操作数指令，也可指定其他的寄存器)；第三条指令把 r1 寄存器的数据存入存储器。在寄存器 - 寄存器结构的代码中，前两条指令把存储器操作数分别取到 r1 和 r2 寄存器中；第三条指令把 r1 和 r2 寄存器中的操作数相加，结果存入 r3 寄存器；第四条指令把 r3 寄存器的数据存入存储器。

Java 虚拟机 JVM 的 Bytecode 指令具有堆栈结构，6502 和 Z80 的指令属于累加器结构，x86 是典型的寄存器 - 存储器结构，而 RISC 类型的 CPU 均具有寄存器 - 寄存器结构，有时也称 Load/Store 结构，其含义是只有 Load 和 Store 指令才访问存储器，计算类指令均使用寄存器操作数。

4.1.5 寻址方式

寻址方式 (Addressing Modes) 是指通过什么样的手段得到操作数。操作数可能在累加器或堆栈中。这时不需要在指令中明确指出，因为它们已经隐含在指令的操作码中了。除此之外，操作数也可能在寄存器或者存储器中，甚至在指令本身中。

[2]玻璃试管一般是用来做化学实验的，不应把它当存钱罐用。

1. 寄存器操作数及立即数寻址

1) 寄存器操作数寻址：操作数在寄存器堆中 (变量)。例如：

```
add r3, r1, r2 ; r3 <-- r1 + r2
```

2) 立即数寻址：立即数 (Immediate) 在指令本身中给出 (常数)。例如：

```
addi r1, r1, -1 ; r1 <-- r1 - 1
```

2. 存储器操作数寻址

1) 直接寻址：直接在指令中给出存储器地址。例如：

```
add r3, r1, [0x1234] ; r3 <-- r1 + Memory[0x1234]
```

2) 寄存器间接寻址：寄存器的内容是存储器地址。例如：

```
add r3, r1, (r2) ; r3 <-- r1 + Memory[r2]
```

3) 偏移量寻址：寄存器的内容与偏移量相加的结果是存储器地址。例如：

```
add r3, r1, 0x1234(r2) ; r3 <-- r1 + Memory[0x1234+r2]
```

4.2 MIPS 指令格式和通用寄存器定义

MIPS 指令系统结构有 MIPS32 和 MIPS64 两种。本书的指令选自 MIPS32[22, 23, 24]。

4.2.1 MIPS 指令格式

MIPS 所有的指令均为 32 位。图 4.5 给出 MIPS 指令的 3 种格式。op 是指令码。

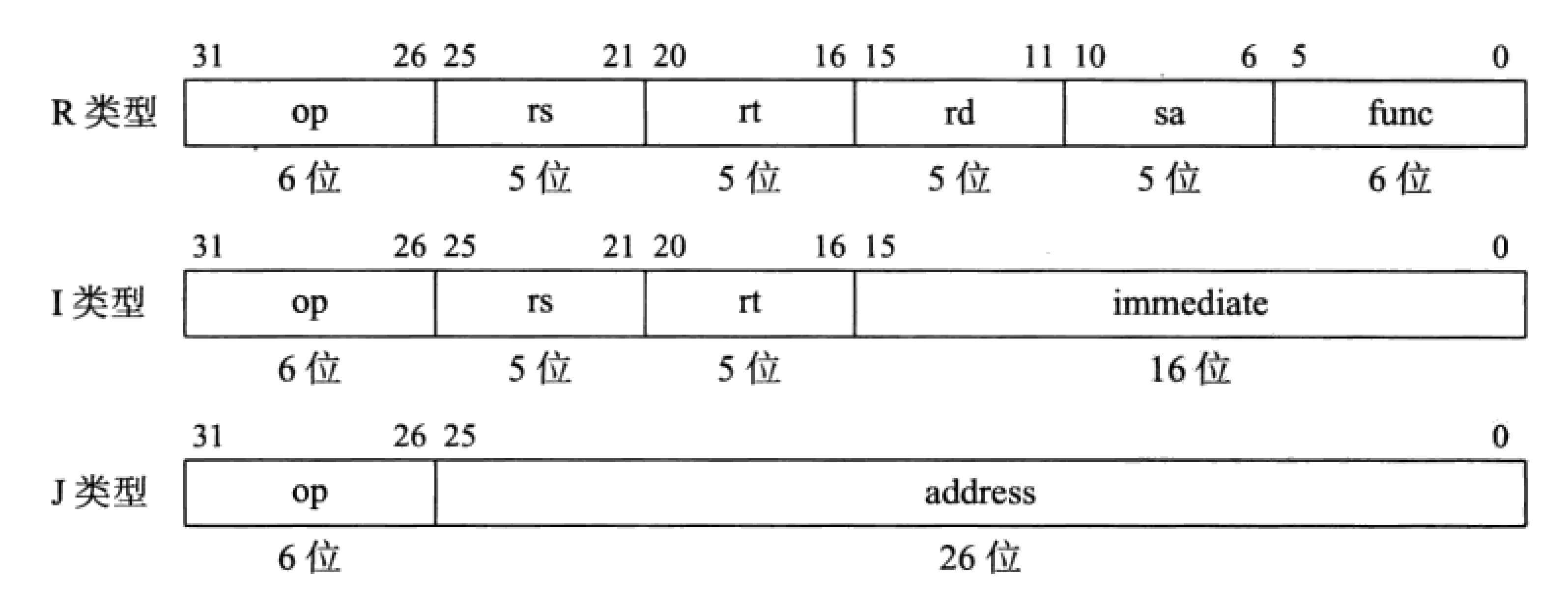


图 4.5 MIPS 指令格式

R 类型指令的 op 为 0，具体操作由 func 指定。rs 和 rt 是源寄存器号，rd 是目的寄存器号。只有移位指令使用 sa 来指定移位位数。I 类型指令的低 16 位是立即数，计算时要把它扩展到 32 位。依指令的不同，有零扩展和符号扩展两种。零扩展是把

32 位的高 16 位置成 0；符号扩展是把高 16 位的每一位置成与立即数最高位相同的值，即保持立即数的正负符号不变。J 类型的指令格式最简单，右边的 26 位是字地址，用于产生跳转的目标地址。

4.2.2 MIPS 通用寄存器

MIPS 指令中的寄存器号 (rs、rt 和 rd) 有 5 位，因此它能访问 $2^5 = 32$ 个寄存器。表 4.3 列出了这 32 个寄存器的名称和它们的用途。

表 4.3 MIPS 通用寄存器

寄存器名	寄存器号	用 途
$zero	0	常数 0
$at	1	汇编器专用
$v0 ~ $v1	2 ~ 3	表达式计算或者函数调用的返回结果
$a0 ~ $a3	4 ~ 7	函数调用参数 1 ~ 3
$t0 ~ $t7	8 ~ 15	临时变量，函数调用时不需要保存和恢复
$s0 ~ $s7	16 ~ 23	函数调用时需要保存和恢复的寄存器变量
$t8 ~ $t9	24 ~ 25	临时变量，函数调用时不需要保存和恢复
$k0 ~ $k1	26 ~ 27	操作系统专用
$gp	28	全局变量指针 (Global Pointer)
$sp	29	堆栈指针 (Stack Pointer)
$fp	30	帧指针 (Frame Pointer)
$ra	31	返回地址 (Return Address)

有两点需要特别注意：第一，0 号寄存器的内容永远是 0。第二，31 号寄存器用来保存返回地址。表 4.3 虽然给出了使用这些寄存器的一些约定，但除了以上两点，这些寄存器并无本质的区别。因此，在以后的描述中，我们不使用带有 $ 的寄存器名，而是直接在 r 后面加寄存器号：r0，r1，···，r31。

4.3 MIPS 指令和 ALU 设计

4.3.1 本书 CPU 可执行的 MIPS 指令

本小节并不介绍所有的 MIPS 指令 (那不是本书的目的)，而只是介绍在本书中的 CPU 能够执行的指令。我们选取若干典型的 MIPS 指令来描述 CPU 逻辑电路的设计方法。表 4.4 列出了其中的整数指令以及它们的格式和意义。

以下我们简要介绍这些指令。前 5 条指令具有相同的格式，只是操作不同：它们分别完成加、减、与、或和异或运算。

```
add/sub/and/or/xor rd, rs, rt   # rd <-- rs op rt
```

表 4.4 20 条 MIPS 整数指令

指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	意 义
add	000000	rs	rt	rd	00000	100000	寄存器加
sub	000000	rs	rt	rd	00000	100010	寄存器减
and	000000	rs	rt	rd	00000	100100	寄存器与
or	000000	rs	rt	rd	00000	100101	寄存器或
xor	000000	rs	rt	rd	00000	100110	寄存器异或
sll	000000	00000	rt	rd	sa	000000	左移
srl	000000	00000	rt	rd	sa	000010	逻辑右移
sra	000000	00000	rt	rd	sa	000011	算术右移
jr	000000	rs	00000	00000	00000	001000	寄存器跳转
addi	001000	rs	rt	immediate			立即数加
andi	001100	rs	rt	immediate			立即数与
ori	001101	rs	rt	immediate			立即数或
xori	001110	rs	rt	immediate			立即数异或
lw	100011	rs	rt	offset			取整数数据字
sw	101011	rs	rt	offset			存整数数据字
beq	000100	rs	rt	offset			相等转移
bne	000101	rs	rt	offset			不等转移
lui	001111	00000	rt	immediate			设置高位
j	000010	address					跳转
jal	000011	address					调用

其中，rs 和 rt 是两个源操作数的寄存器号，rd 是目的寄存器号。注意它们在汇编指令及指令格式中的位置。一般我们用分号表示注释部分，但 MIPS 汇编语言使用 #。

```
sll/srl/sra rd, rt, sa   # rd <-- rt shift sa
```

这是 3 条移位指令 (Shift Left / Right Logical /Arithmetic)，5 位的 sa (Shift Amount) 指定移位的位数。我们已经在第 2 章详细描述了这 3 种移位操作并给出了相应的 Verilog HDL 代码。

```
lui rt, imm # rt <-- imm << 16
```

lui (Load Upper Immediate) 指令把 16 位立即数 imm 左移 16 位，存入 rt 寄存器。它与 ori 指令合作，可以为一个 32 位的寄存器赋任意值：lui 赋高 16 位，ori 赋低 16 位。

```
addi rt, rs, imm  # rt <-- rs + imm (符号扩展)
```

addi (Add Immediate) 是立即数的加法指令。注意目的寄存器号是 rt，立即数要符号扩展到 32 位。因为是符号扩展，因此 MIPS 指令系统中没有类似于 subi 这样的指令。

```
andi/ori/xori rt, rs, imm  # rt <-- rs op imm (零扩展)
```

这 3 条是逻辑操作指令 (And / Or / Xor Immediate)，因此立即数要零扩展。

```
lw rt, offset(rs) # rt <-- memory[rs + offset]
```

lw (Load Word) 是一条取存储器字的指令。寄存器 rs 的内容与符号扩展的 offset 相加，得到存储器地址。从存储器取来的数据存入 rt 寄存器。注意，offset 就是前面讲的立即数。

```
sw rt, offset(rs) # memory[rs + offset] <-- rt
```

sw (Store Word) 是一条存字的指令，与 lw 方向相反，把 rt 寄存器的内容放入存储器。存储器地址的计算与 lw 相同。

```
beq rs, rt, label # if (rs == rt) PC <-- label
```

beq (Branch on Equal) 是一条条件转移指令。当寄存器 rs 的内容与寄存器 rt 的内容相等时，转移到 label。如果程序计数器 PC 是 beq 指令的地址，则 label = PC + 4 + offset << 2。offset 左移两位导致 PC 的最低两位永远是 0，这是因为 PC 是字节地址而一条指令要占 4 个字节。offset 是要符号扩展的，因此 beq 能实现向前和向后两种转移。

```
bne rs, rt, label # if (rs != rt) PC <-- label
```

与 beq 类似，但 bne (Branch on Not Equal) 是在两个寄存器的内容不相等时转移。

```
j target # PC <-- target
```

j (Jump) 是一条跳转指令。target 是跳转的目标地址，32 位，由 3 部分组成：最高 4 位来自于 PC + 4 的高 4 位，中间 26 位是指令中的 address，最低两位为 0。这条指令在生成目标地址时不需要任何电路进行计算，只需把 3 部分地址拼接起来就行。以下的两条指令也不需要计算。

```
jal target # r31 <-- PC + 8; PC <-- target
```

jal (Jump and Link) 指令与 j 类似，但要把返回地址保存在 r31 中。即 jal 是子程序调用指令。jal 下一条指令的地址是 PC + 4，为什么返回地址是 PC + 8? 这是因为 MIPS 指令系统实现流水线的延迟转移功能，详见第 8 章。注意，寄存器号 31 是约定好的，该号码并不出现在指令中。因此在设计电路时，应当由硬件为 jal 指令产生这个号码。

```
jr rs # PC <-- rs
```

jr (Jump Register) 也是一条跳转指令，它把 rs 寄存器的内容写入 PC。如果指定 rs 为 31，则 jr 是从子程序返回的指令。

表 4.5 是我们的 CPU 能执行的异常处理、TLB 管理及浮点指令。有关这些指令的意义及实现，我们将在以后相关章节描述。

表 4.5 MIPS 中断和异常处理、TLB 管理以及浮点运算指令

指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	意 义
syscall	000000	00000	00000	00000	00000	001100	系统调用
eret	010000	10000	00000	00000	00000	011000	从异常返回
tlbwi	010000	10000	00000	00000	00000	000010	写指定的 TLB 项
tlbwr	010000	10000	00000	00000	00000	000110	写随机的 TLB 项
mfc0	010000	00000	rt	rd	00000	000000	取控制字
mtc0	010000	00100	rt	rd	00000	000000	存控制字
lwc1	110001	rs	ft	offset			取浮点数据字
swc1	111001	rs	ft	offset			存浮点数据字
add.s	010001	10000	ft	fs	fd	000000	单精度浮点加
sub.s	010001	10000	ft	fs	fd	000001	单精度浮点减
mul.s	010001	10000	ft	fs	fd	000010	单精度浮点乘
div.s	010001	10000	ft	fs	fd	000011	单精度浮点除
sqrt.s	010001	10000	00000	fs	fd	000100	单精度浮点开方

4.3.2 ALU 设计

ALU 是负责运算的电路。综合表 4.4 的指令，ALU 需实现以下的运算：ADD (加)、SUB (减)、AND (与)、OR (或)、XOR (异或)、LUI (置高位立即数)、SLL (逻辑左移)、SRL (逻辑右移) 和 SRA (算术右移)。

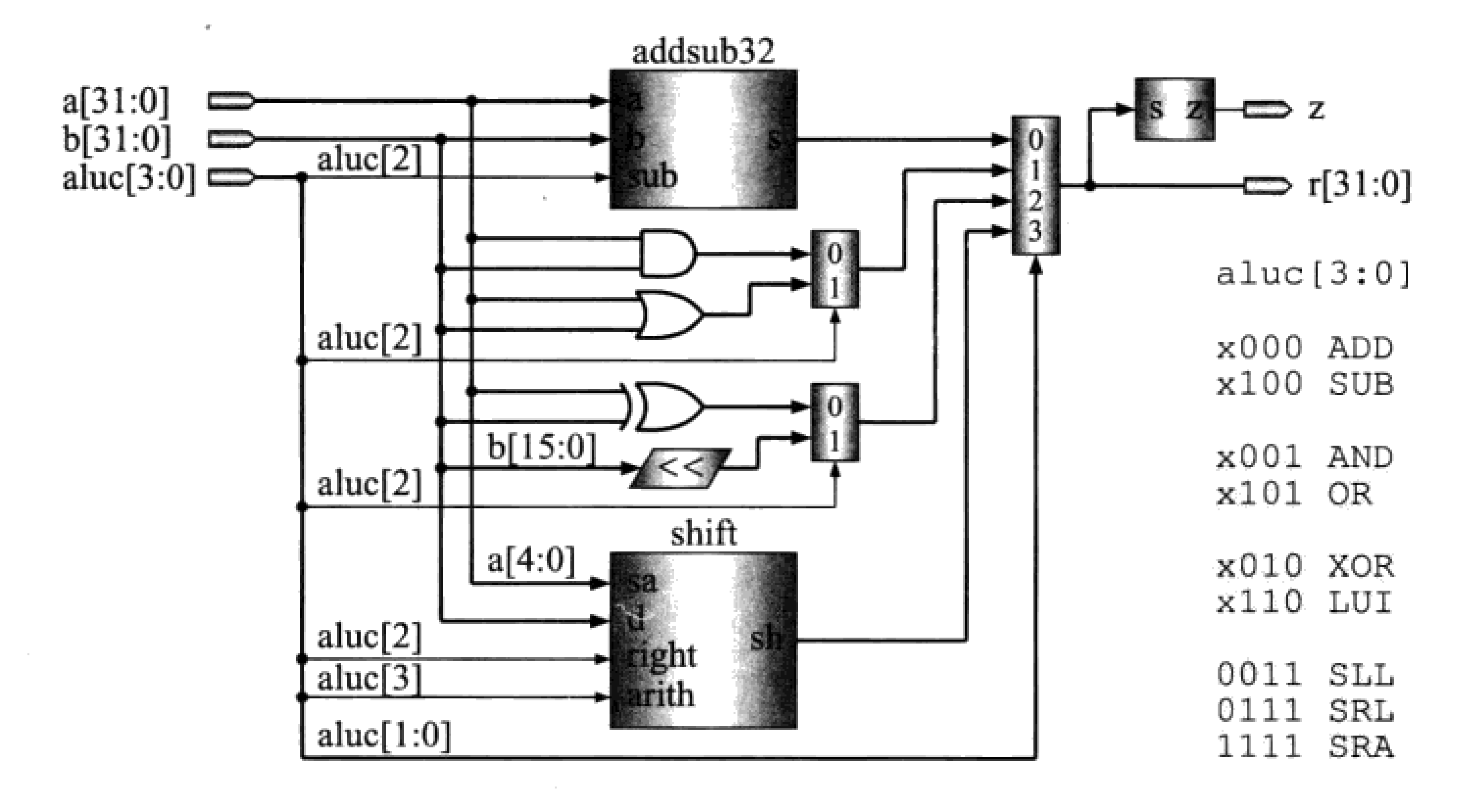


图 4.6 ALU 的逻辑电路图

图 4.6 是 ALU 的逻辑电路图。a[31:0] 和 b[31:0] 是两个 32 位的输入，aluc[3:0] 是 ALU 的操作控制码，r[31:0] 是 ALU 的 32 位输出，z 是一位零标志。当 r[31:0] 为

0 时，z 输出 1。

ALU 设计的基本想法是准备好所有的运算电路，然后用多路器进行选择。多路器的选择信号就是 ALU 的操作控制码。实际上，图 4.6 是先画好了电路，才定下了操作控制码 aluc。

以下是 ALU 的 Verilog HDL 代码，用到了以下三个模块：addsub32、shift 和 mux4x32。后两个模块已在第 2 章给出。

```
module alu (a,b,aluc,r,z);
    input [31:0] a,b;                                    // aluc[3:0]
    input [3:0]  aluc;                                   //
    output [31:0] r;                                     // x 0 0 0  ADD
    output z;                                            // x 1 0 0  SUB
    wire [31:0] d_and = a & b;                           // x 0 0 1  AND
    wire [31:0] d_or  = a | b;                           // x 1 0 1  OR
    wire [31:0] d_xor = a ^ b;                           // x 0 1 0  XOR
    wire [31:0] d_lui = {b[15:0],16'h0};                 // x 1 1 0  LUI
    wire [31:0] d_and_or  = aluc[2]? d_or  : d_and;      // 0 0 1 1  SLL
    wire [31:0] d_xor_lui = aluc[2]? d_lui : d_xor;      // 0 1 1 1  SRL
    wire [31:0] d_as,d_sh;                               // 1 1 1 1  SRA
    addsub32 as32 (a,b,aluc[2],d_as);
    shift shifter (b,a[4:0],aluc[2],aluc[3],d_sh);
    mux4x32 select (d_as,d_and_or,d_xor_lui,d_sh,aluc[1:0],r);
    assign z = ~|r;
endmodule
```

上述代码中标志位 z 的赋值语句的意思是把 r 的所有 32 位或起来再取反。

以下是 addsub32 模块的 Verilog HDL 代码，它实现 32 位的加减运算。由于 $a - b = a + (-b) = a + \bar{b} + 1$，所以 addsub32 可以使用加法电路 cla32 实现。cla32 的代码已在第 3 章给出了。

```
module addsub32 (a, b, sub, s);
    input  [31:0] a, b;
    input         sub;
    output [31:0] s;
    cla32 as32 (a, b^{32{sub}}, sub, s);
endmodule
```

图 4.7 是 ALU 的仿真波形，其中 aluc 用二进制表示，其余是十六进制。测试的次序是 ADD、SUB、AND、OR、XOR、LUI、SLL、SRL 和 SRA，其中 SRA 测试了数据最高位为 1 和 0 两种情况。当 r 为 0 时，z 应该输出 1，我们没有测试。

我们将从第 5 章开始，使用这个 ALU 来设计各种不同的 CPU。因此建议读者仔细检查信号的输入值和输出值，看输出值是否与我们期待的一样，同时也加深对 ALU 设计方法的理解。

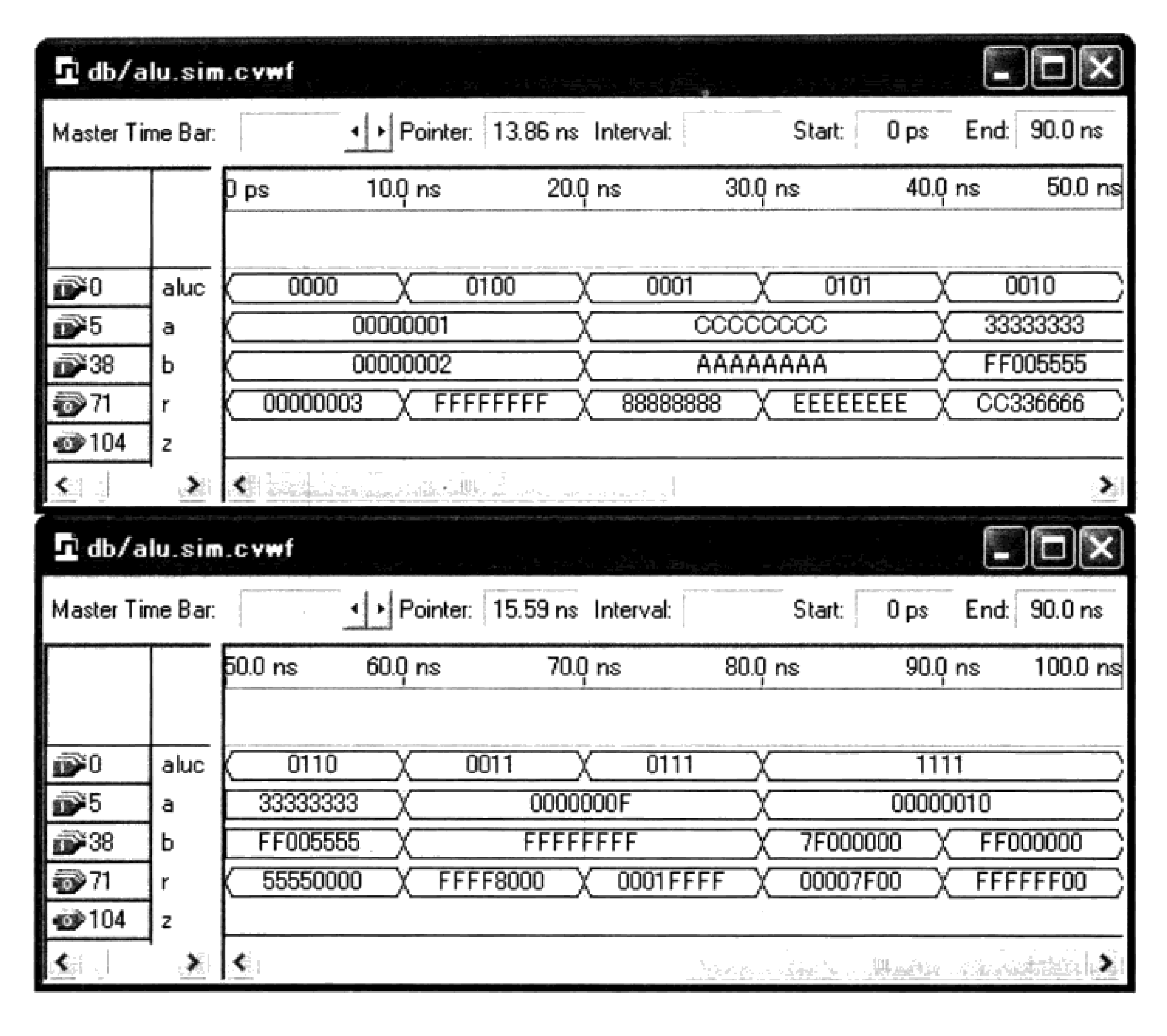


图 4.7 ALU 仿真波形

4.4 习题

1. 使用功能描述风格的 Verilog HDL (casex) 重新设计 ALU 并仿真。
2. 在 ALU 中增加一位标志位 v (输出)，判断带符号数计算时的溢出。
3. 试实现 sla (Shift Left Arithmetic) 指令 (算术左移、保持正负不变)。
4. 参照表 4.4，试把以下汇编程序转换成二进制代码，假设地址从 0 开始。

```
main:   lui  r1, 0          # address of data[0]
        ori  r4, r1, 80     # address of data[0]
        addi r5, r0,  4     # counter
call:   jal  sum            # call function
        sw   r2, 0(r4)      # store result
        lw   r9, 0(r4)      # check sw
        sub  r8, r9, r4     # sub: r8 <-- r9 - r4
        addi r5, r0,  3     # counter
loop2:  addi r5, r5, -1     # counter - 1
        ori  r8, r5, 0xffff # zero-extend: 0000ffff
```

```
        xori r8, r8, 0x5555   # zero-extend: 0000aaaa
        addi r9, r0, -1       # sign-extend: ffffffff
        andi r10, r9, 0xffff  # zero-extend: 0000ffff
        or   r6, r10, r9      # or:  ffffffff
        xor  r8, r10, r9      # xor: ffff0000
        and  r7, r10, r6      # and: 0000ffff
        beq  r5, r0, shift    # if r5 = 0, goto shift
        j    loop2            # jump loop2
shift:  addi r5, r0, -1       # r5   = ffffffff
        sll  r8, r5, 15       # <<15 = ffff8000
        sll  r8, r8, 16       # <<16 = 80000000
        sra  r8, r8, 16       # >>16 = ffff8000 (arith)
        srl  r8, r8, 15       # >>15 = 0001ffff (logic)
finish: j    finish           # dead loop
sum:    add  r8, r0, r0       # sum
loop:   lw   r9, 0(r4)        # load data
        addi r4, r4,  4       # address + 4
        add  r8, r8, r9       # sum
        addi r5, r5, -1       # counter - 1
        bne  r5, r0, loop     # finish?
        sll  r2, r8, 0        # move result to v0
        jr   r31              # return
```

5. 调查以下 CPU 的指令系统，包括寄存器 (累加器) 结构和寻址方式，并比较它们的特点：8086、Z80、6800、6502、SPARC、ARM。
6. 使用 gcc -S my_program.c 可以把高级语言程序转换成汇编程序。假设你现在使用的不是 MIPS CPU 的机器而又想生成 MIPS 汇编程序，有办法吗？有，去网上下载一个交叉编译程序。试试看。

第 5 章　单周期 CPU 及其 Verilog HDL 设计

目前的计算机都属于“同步”计算机。所谓同步计算机是指在计算机系统中有一个时钟 (Clock)，计算机所有的动作都以这个时钟为基准。见图 5.1 单周期 CPU 的波形，我们把时钟的电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期 (Clock Cycle)。

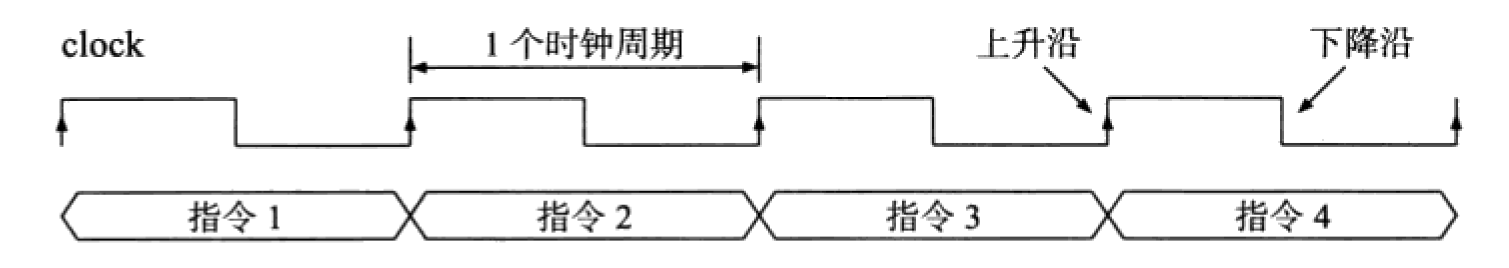


图 5.1　时钟周期和单周期 CPU 指令的执行

单周期 CPU 指的是一条指令的执行在一个这样的时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个周期。

从本章开始正式介绍 CPU 的设计方法。当然，本章只涉及单周期 CPU，以后各章陆续介绍比单周期 CPU 性能要好的其他类型的 CPU 的设计方法。本章是基础。

5.1　执行一条指令所需的硬件电路

计算机程序由一条条指令及数据构成。通常程序存于硬盘中。当计算机执行一个程序时，首先由操作系统把要执行的程序从硬盘调入内存，然后 CPU 从内存读出指令开始执行。注意操作系统本身也是程序。

我们的目的是要设计 CPU 的硬件电路，使其能够从存储器中读取一条条指令并执行指令所描述的操作。从存储器读取指令的动作一般与指令本身的意义无关，我们可以对所有的指令以同样的方法从存储器中取来。而执行指令则与指令本身的意义密切相关，因此最重要的是首先搞清楚 CPU 要执行的每一条指令的意义。

第 4 章的表 4.4 列出了我们要设计的 CPU 能够执行的 20 条整数指令。依指令格式分类，有寄存器格式、立即数格式以及跳转格式 3 种。依指令意义分类，有 4 种：计算类型、访问存储器类型、条件转移类型和无条件跳转类型[1]。我们将对每条指令执行时所需的硬件电路分别加以说明，然后给出一个整体 CPU 电路。

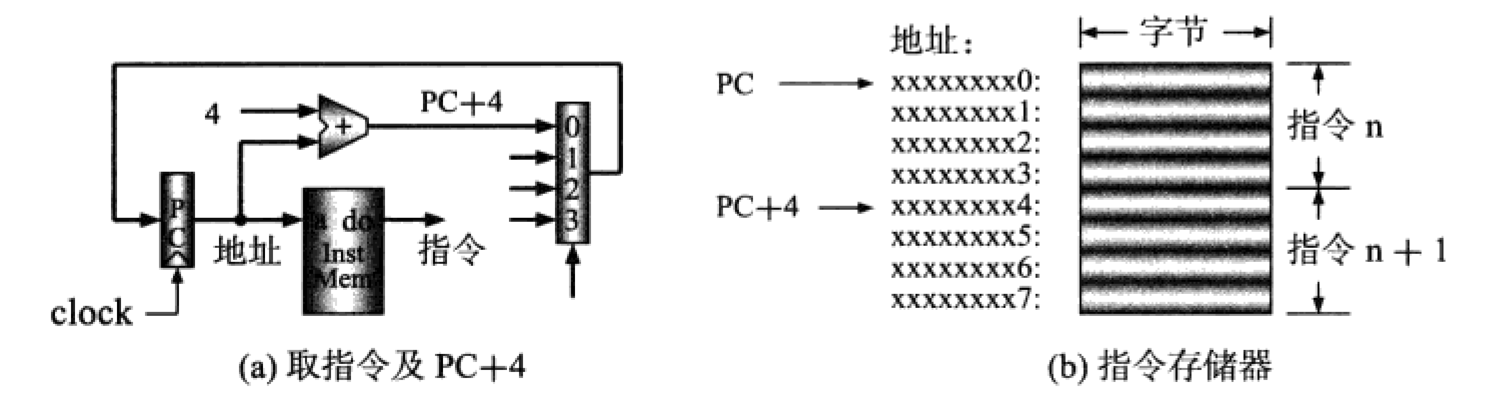
5.1.1　与取指令有关的电路

指令在存储器中。CPU 要执行它必须首先把它从存储器中读出来，然后才能知道指令究竟要干什么。CPU 取指令时把程序计数器 (Program Counter，PC) 中的内容作为存储器地址，根据它来访问存储器，从 PC 值指定的存储器单元中取来一条 32

[1]单周期 CPU 忽略了 MIPS 转移类指令的延迟转移特性，我们将在流水线 CPU 中实现延迟转移。

位指令。如果取来的指令执行时没有引起转移，PC 的值要加 4；如果转移，要把转移目标地址写入 PC，以便在下一个时钟周期取出下一条指令。

图 5.2(a) 所示的是与取指令有关的电路，其中的 PC 是一个简单的 32 位寄存器，由 32 个 D 触发器构成。指令存储器 (图中的 Inst Mem) 的输入端 a 是地址 (Address)、输出端 do 是数据输出 (Data Out)，即指令。图中的加法器专供 PC + 4 使用，它的输出接到多路器的一个输入端。如果取来的指令没有引起转移或跳转，则选择 PC + 4，在时钟上升沿处将其打入 PC。多路器的其他输入信号将在稍后描述。



(a) 取指令及 PC+4　　　　(b) 指令存储器

图 5.2　取指令时用到的硬件电路

为什么 PC 要加 4 而不是加 1 呢？这和存储器地址的约定有关。一般来讲，存储器的容量以字节为单位表示，例如 4G 字节，因此存储器的地址通常也是存储器的字节地址。我们知道，一个字节有 8 位二进制位，而一条指令有 32 位，要用 4 个字节，所以要加 4，见图 5.2(b)。注意，32 位 PC 能访问 $2^{32} = 4G$ 字节的存储器，但我们的设计只使用了少量的存储器，因此 PC 的高位没被使用。

5.1.2　寄存器计算类型指令执行时所需电路

寄存器计算类型的指令有 add、sub、and、or、xor、sll、srl 和 sra。图 5.3 所示的是执行指令 add、sub、and、or 和 xor 时所需的硬件电路。这 5 条指令除了运算不同，其他动作均相同。

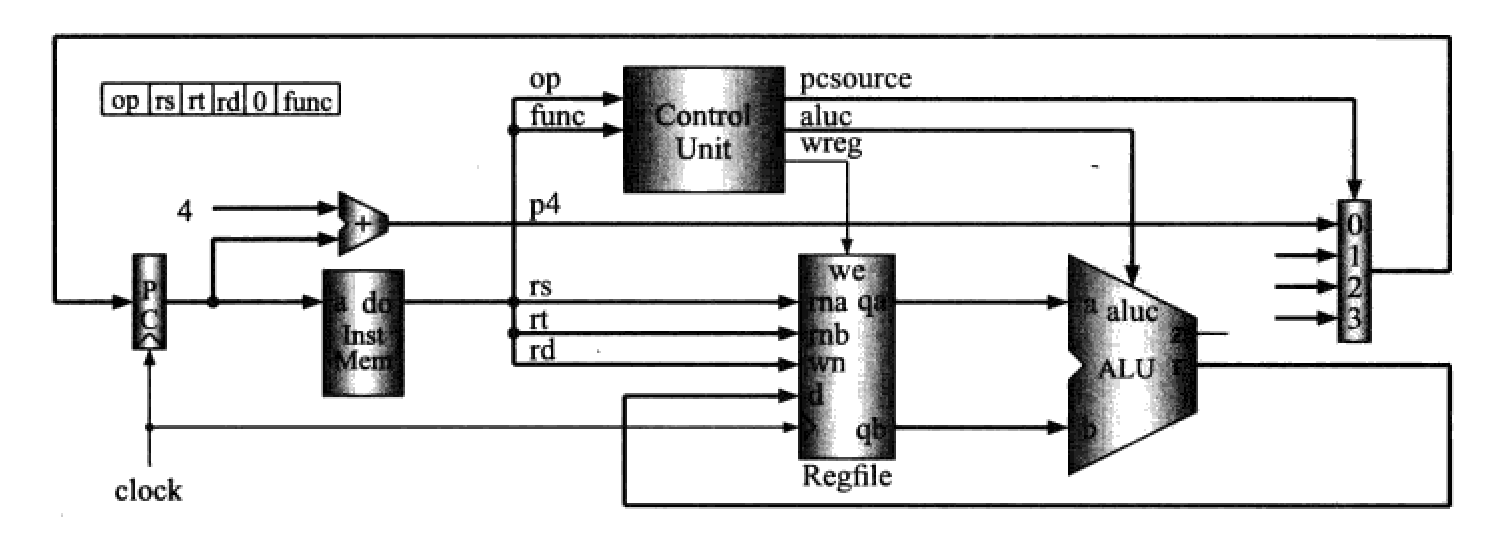


图 5.3　执行 add、sub、and、or 和 xor 指令所需的电路

因为它们都属于寄存器格式的指令，它们的操作码 op 均为 0，具体操作由功能

码 func 指定，见表 4.4。大多数 MIPS 指令属于三操作数指令。指令格式中的 rs 和 rt 是两个 5 位的寄存器号，由它们从寄存器堆 (图中的 Regfile) 中读出两个 32 位的数据。由于寄存器号有 5 位，所以能从 $2^5 = 32$ 个寄存器中选出一个。32 个寄存器合在一起称为寄存器堆 (Register File)。有关寄存器堆的设计方法将在后面讲解。从寄存器堆读出的两个 32 位数据分别被送到 ALU 的 a 和 b 的输入端。

具体的计算由 ALU 完成。ALU 的计算控制码 aluc (ALU Control) 由控制部件 (Control Unit) 产生。我们已经在第 4 章描述了 ALU 的设计。这里的控制部件是简单的组合电路，输入信号是指令的操作码 op 和功能码 func，输出信号有 3 个，它们分别是 ALU 操作控制码 aluc、计算结果是否写入寄存器堆的控制信号 wreg (Write Register File) 和下一条指令的地址选择信号 pcsource (PC Source)。有关控制部件的具体设计方法将在后面描述。ALU 的计算结果要被写入寄存器堆。到底写入 32 个寄存器中的哪一个，由 5 位目的寄存器号 rd 指定。

图 5.4 所示的是执行移位指令 sll (Shift Left Logical)、srl (Shift Right Logical) 和 sra (Shift Right Arithmetic) 时所需的硬件电路。同样是寄存器计算类型的指令，sll、srl 和 sra 这 3 条指令的格式则有些怪怪的，见表 4.4。它们使用 5 位的寄存器号 rt 从寄存器堆中读出 32 位数据，然后把这 32 位数据左移或右移 (逻辑右移或算术右移)。移位位数由指令中的 5 位 sa (Shift Amount) 决定。移位后的结果写入由 5 位 rd 指定的寄存器中。寄存器号 rs 并没有使用，按 MIPS 的规定，把这 5 位设置为 0。

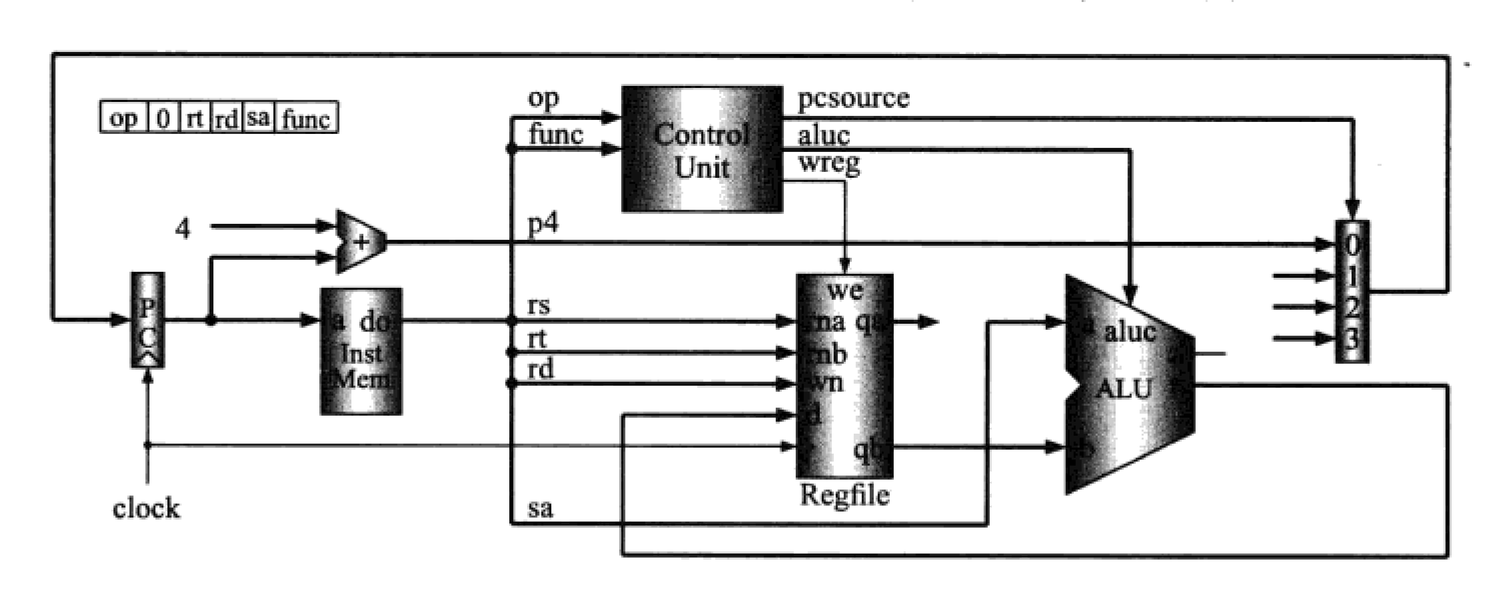


图 5.4 执行 sll、srl 和 sra 指令所需的电路

执行这 3 条移位指令所需的硬件电路与图 5.3 所示的电路略有不同：接入 ALU 输入端 a 的是指令中的 sa；而寄存器堆的输出端 qa 并没被使用。由于 sa 只有 5 位，而 ALU 的输入端 a 需要 32 位，我们把 sa 放在最右，左边的 27 位随便放什么都行，因为我们在设计 ALU 时只使用低 5 位进行移位操作。电路的其余部分与图 5.3 相同。由于 ALU 的输入端 a 的数据来源有两个 (一个来自寄存器堆的 qa，另一个来自指令中的 sa)，我们必须根据取来的指令，从这两个数据源中选出一个。使用什么来选？当然是二选一多路器 (多路器真是好东西啊)。二选一多路器的两个数据输入端分别接两个数据源，选择信号由控制部件产生。

5.1.3 立即数计算类型指令执行时所需电路

立即数计算类型的指令包括 addi、andi、ori、xori 和 lui，仅由 op 区分。它们的共同特点是 ALU 操作数 b 来自于指令中的立即数，见图 5.5。

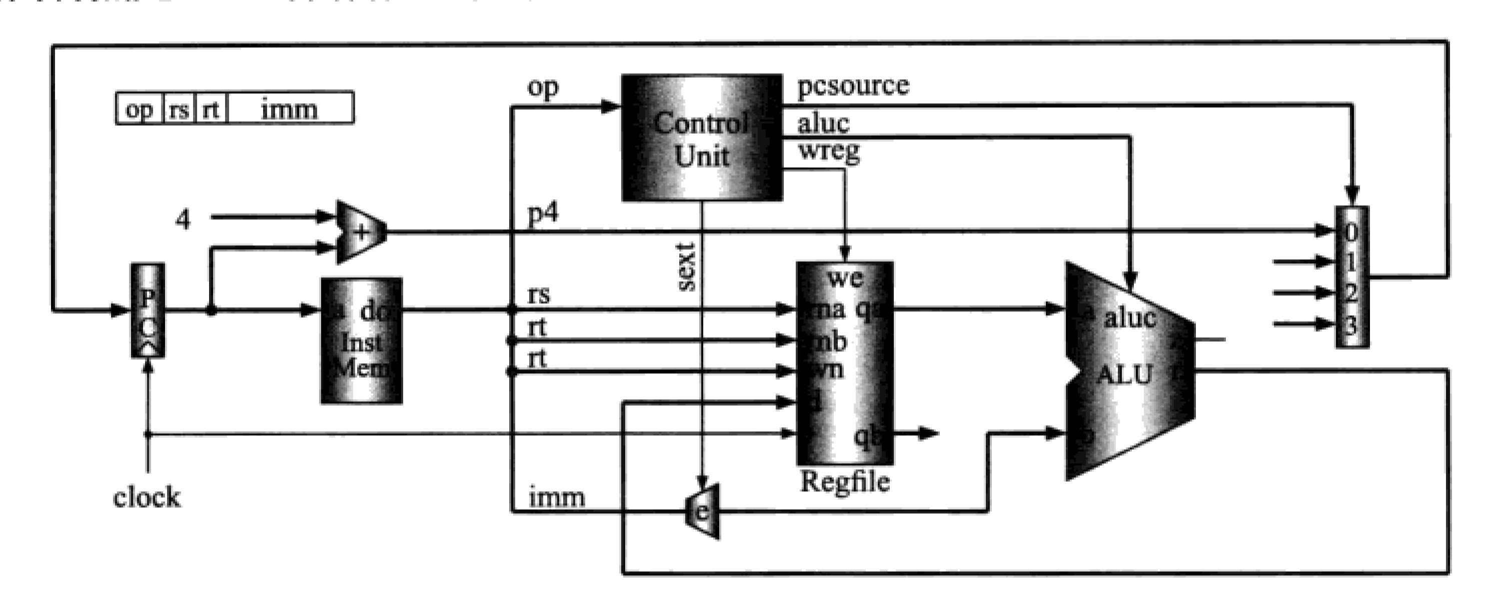


图 5.5 执行 addi、andi、ori、xori 和 lui 指令所需的电路

指令 lui 比较特殊，它只需要一个操作数 (立即数)。ALU 将其左移 16 位，结果保存到由 rt 指定的寄存器中。其他指令均需两个操作数。操作数 a 来自于 rs 寄存器。操作数 b 来自于指令中的立即数。立即数为 16 位，送到 ALU 之前需要扩展成 32 位。算术运算指令 addi 进行符号扩展，逻辑运算指令 andi、ori 和 xori 进行零扩展。图中的控制信号 sext (Sign Extend) 为 1 时符号扩展，否则零扩展。组件 e 即为立即数扩展电路。与寄存器格式的指令不同，立即数格式的指令把计算结果写入由 rt 指定的寄存器中 (寄存器格式的指令的目的寄存器号是 rd)。

5.1.4 访问存储器类型指令执行时所需电路

访问存储器的指令有两条：lw (Load Word) 指令从数据存储器中读数据，sw (Store Word) 指令往存储器中写数据。两条指令计算存储器地址的方法是把两个数相加：一个数是使用 rs 从寄存器堆中读出的，另一个数是把指令中的 16 位立即数进行符号扩展得到的。两个数的相加由 ALU 完成，相加结果作为存储器的地址使用。lw 指令把从存储器中读出的数据写入由 rt 指定的寄存器中；sw 指令把 rt 寄存器的数据写入存储器。

图 5.6 所示的是执行 lw 和 sw 指令所需的硬件电路。图中的符号扩展控制信号 sext 输出 1。是否写入数据存储器 (图中的 Data Mem) 由控制信号 wmem (Write Memory) 指定：为 1 时写入，为 0 时不写。当指令是 lw 时，wreg = 1，wmem = 0；当指令是 sw 时，wmem = 1，wreg = 0。

由于我们这里讨论的是单周期 CPU 的设计，我们需要两个分开的存储器模块：指令存储器和数据存储器。如果使用一个存储器模块并且存储器模块只有一个访问端口，我们无法在一个时钟周期既读取指令又访问数据。在实际的 CPU 中，CPU 芯

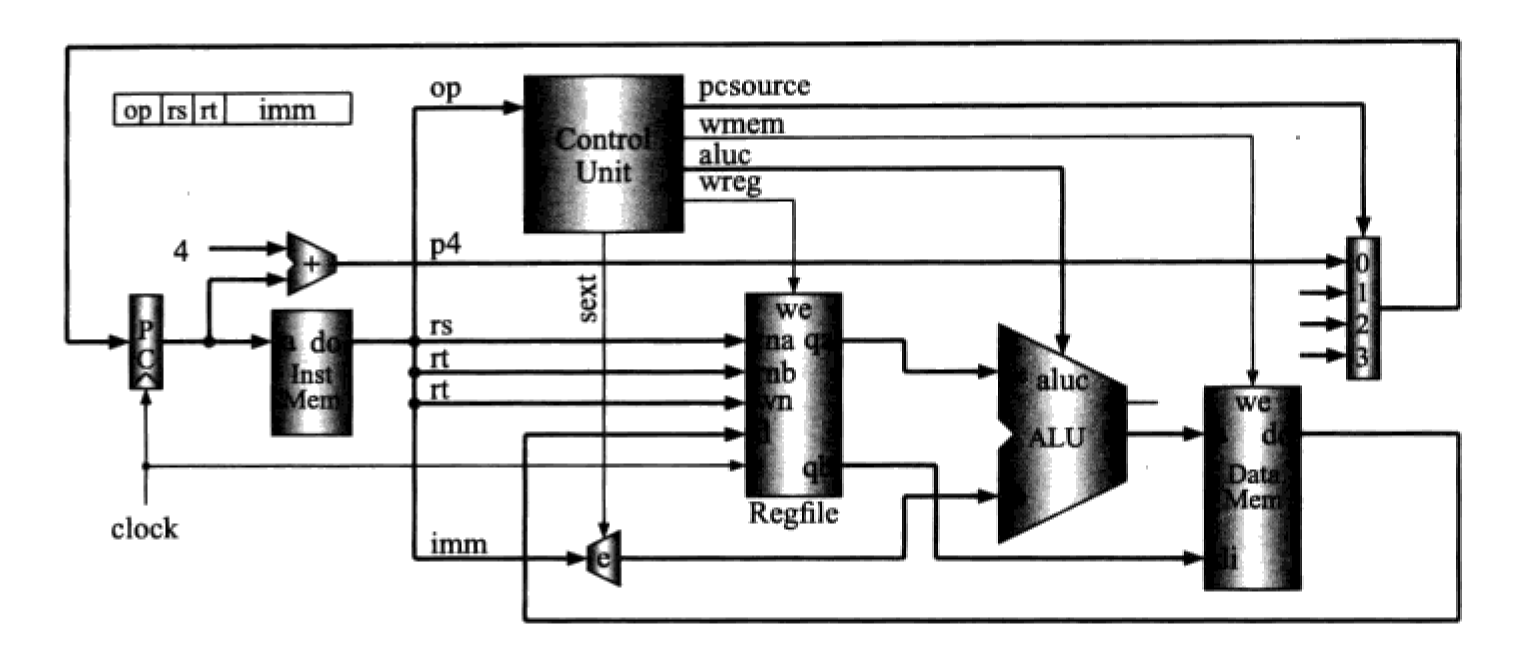


图 5.6 执行 lw 和 sw 指令所需的电路

片内部都设计有分开的指令 Cache 和数据 Cache。后面将讨论有关 Cache 的原理和设计，我们这里暂且"认为"两个存储器模块是两个 Cache 模块。

5.1.5 条件转移类型指令执行时所需电路

到现在为止，在我们给出的电路中，pcsource 信号都是设置为 00，即选择 PC + 4 作为下一条指令的地址。条件转移 (Conditional Branch) 类型的指令则不同，有可能引起向其他地方的转移。这样的指令有 beq (Branch on Equal) 和 bne (Branch on Not Equal) 两条。

执行 beq 和 bne 指令所需的硬件电路如图 5.7 所示。首先使用 rs 和 rt 从寄存器堆读出两个数据，由 ALU 比较它们是否相等，然后决定是否转移。beq 指令是相等时转移，bne 是不等时转移。转移时应使 pcsource = 01，选择转移的目标地址；不转移时应使 pcsource = 00，选择 PC + 4。目标地址的计算由一个专门的加法器完成。加法器的两个输入端分别是 PC + 4 和符号扩展的立即数再左移两位的数据。我们也称这时的立即数为偏移量，使用它做 PC 相对转移。

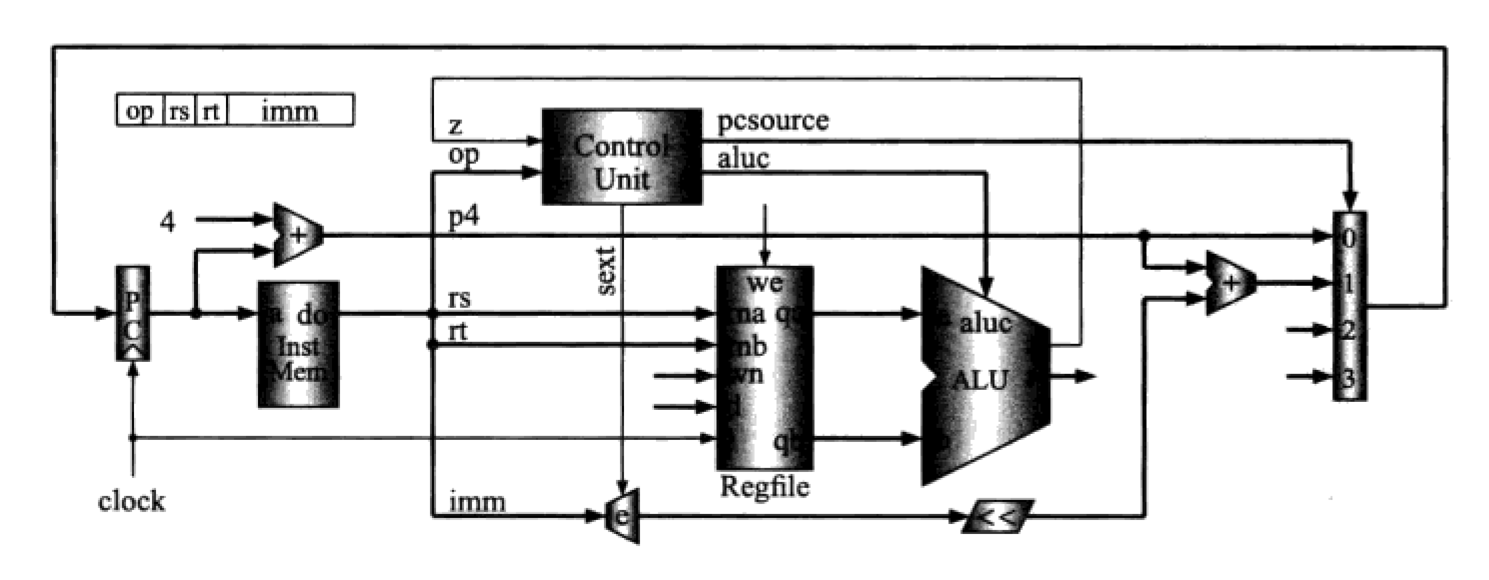


图 5.7 执行 beq 和 bne 指令所需的电路

判断两个数据是否相等可以用减法操作实现：如果相等，减法结果是 0，ALU 的标志位 z = 1；否则 z = 0。我们也可以使用异或操作来判断两个数是否相等：如果两个 32 位数相等，按位异或的结果是 32 位 0，这时 z = 1；否则 z = 0。由于我们设计的 ALU 能完成这两种计算，因此可以任选一种来判断两个数是否相等。

由于一条指令占用 4 个字节，字节地址的 PC 的低两位为 0，所以 PC 相对转移时使用的偏移量的最低两位按道理也应是 0。但把这样的偏移量放在指令的低 16 位有些浪费，因此 beq 和 bne 指令中低 16 位存放的是“字偏移量”而不是“字节偏移量”。使用字偏移量与字节地址 PC + 4 相加时，要把字偏移量左移两位后再相加。左移两位的操作不需要任何硬件电路，只要通过适当的连线即可完成。

5.1.6 跳转和子程序调用及返回类型指令执行时所需电路

条件转移指令能改变程序执行的流程。同样，无条件跳转 (Unconditional Jump) 指令也能，而且是无条件的。20 条指令的最后 3 条是 j (Jump)、jal (Jump and Link) 和 jr (Jump Register)。指令 j 实现无条件跳转，指令中的 26 位地址左移两位变成 28 位，再与 PC + 4 的高 4 位拼接在一起，构成 32 位转移地址，写入 PC。指令 jal 是调用子程序的指令。它除了完成与 j 相同的任务之外，还要把 PC + 4 写入寄存器 r31，即保存返回地址[2]。指令 jr 把从 rs 指定的寄存器中读出的内容写入 PC，可以用于从子程序返回 (rs = 31)。

图 5.8 所示的是 j 指令执行时所需的硬件电路。这可能是 MIPS 指令系统中最简单的指令了。它既不使用 ALU，也不使用寄存器堆。左移两位也是通过连线实现，不需要任何逻辑电路。由于是无条件转移，控制部件输出 pcsource = 11 (3)。

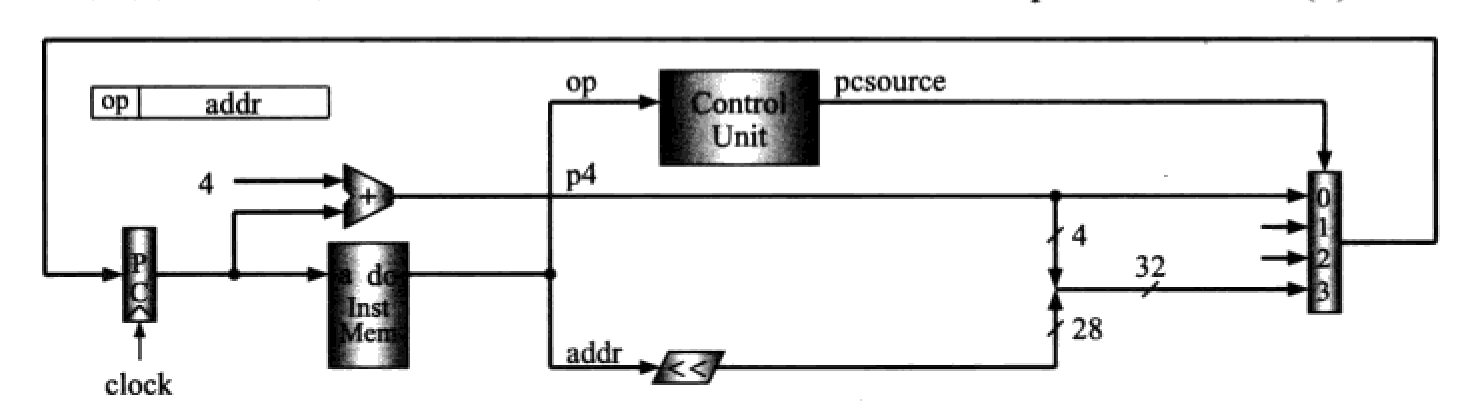


图 5.8 执行 j 指令所需的电路

图 5.9 所示的是 jal 指令执行时所需的硬件电路。该指令除了跳转之外，还要把返回地址保存在 r31 寄存器中，因此我们令 wreg = 1 并且把 5 位二进制数的 11111 送入寄存器堆的 wn 输入端。

图 5.10 所示的是 jr 指令执行时所需的硬件电路：它实现寄存器间接转移：从寄存器堆中读出由 rs 指定的寄存器的内容并把它写入 PC (pcsource = 10)。如果寄存器号是 31，则可实现从子程序返回。

---

[2]MIPS 指令系统考虑到流水线延迟转移的因素，把 PC + 8 作为返回地址保存在 r31。我们这里暂且保存 PC + 4，但要注意这样一来就不与 MIPS 的指令兼容了。

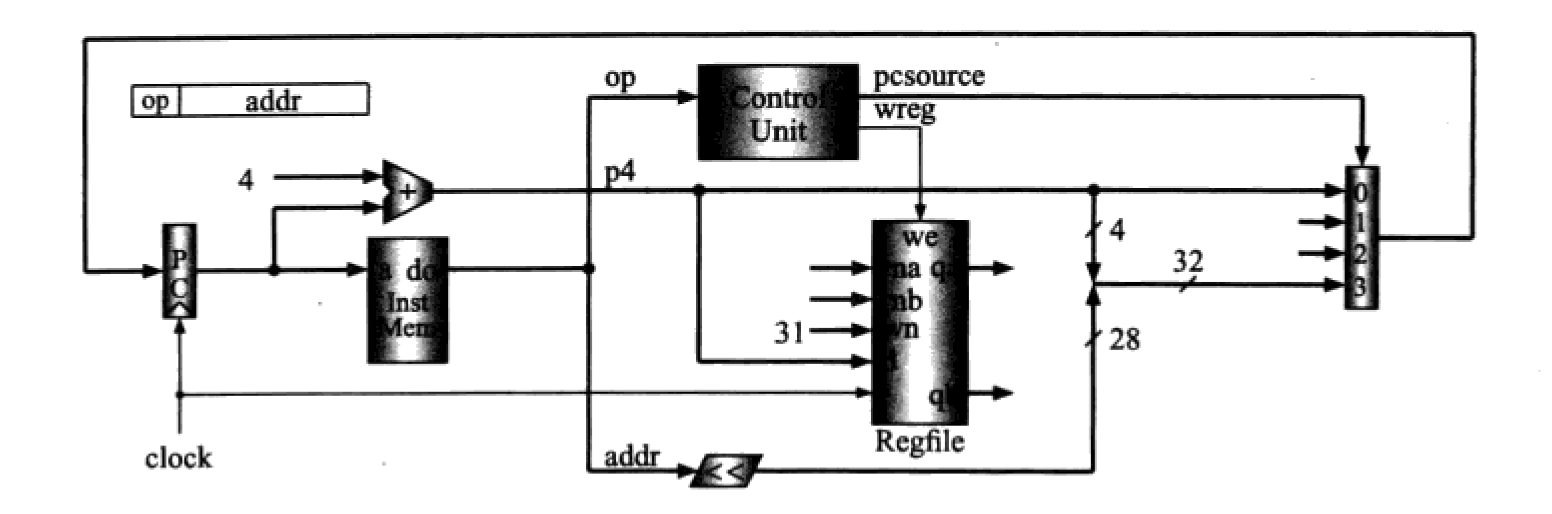


图 5.9 执行 jal 指令所需的电路

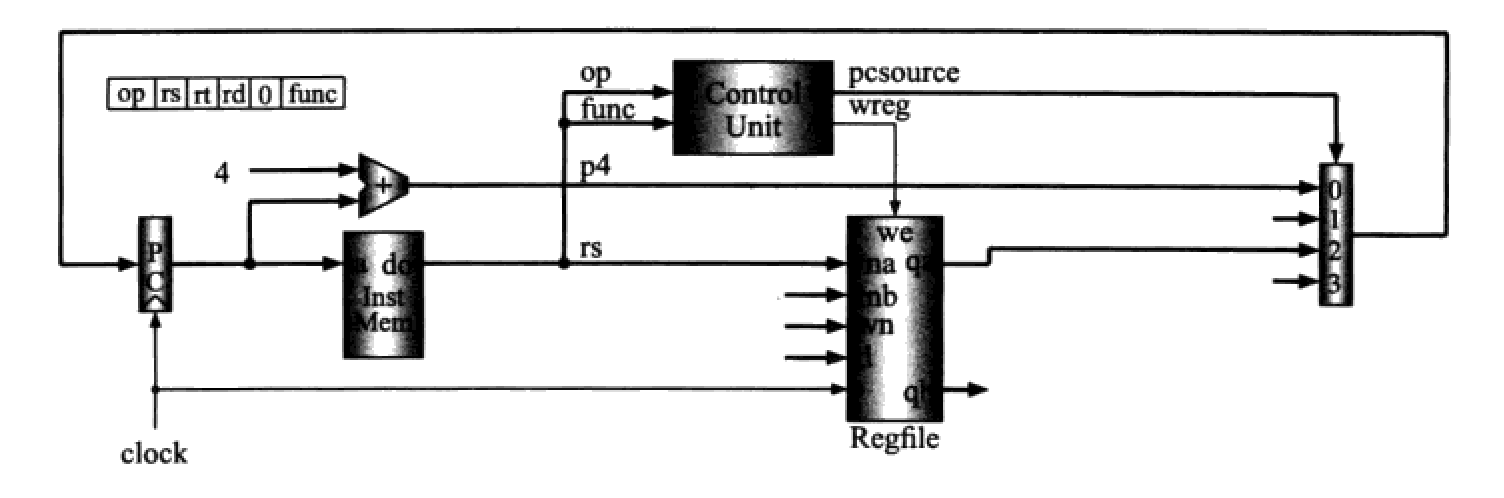


图 5.10 执行 jr 指令所需的电路

所有的 20 条指令在执行时所需的硬件电路都分别介绍完了。从这样的零敲碎打还不足以看出一个整体 CPU 的全貌，有些部件的输入还存在着冲突。稍后我们将介绍如何把它们有机地整合在一起，以指令为本，构建一个和谐的 CPU。在此之前，我们要详细讨论寄存器堆的设计方法。

5.2 寄存器堆设计

MIPS 指令格式中的寄存器号是 5 位，这意味着指令可以访问 $2^5 = 32$ 个寄存器。这样的一堆寄存器"堆在一起"构成一个寄存器堆 (Register File)。图 5.11 所示的是寄存器堆的电路符号。它有两个读端口和一个写端口。每个端口都有一个 5 位的寄存器号，用于指定一个寄存器；还有一个 32 位的数据端，用于读写数据。

我们首先描述寄存器堆的硬件结构，然后给出两个版本的 Verilog HDL 实现代码：一个是结构描述风格的代码，另一个是功能描述风格的代码。

5.2.1 寄存器堆的硬件电路设计

MIPS32 体系结构的 CPU 中有 32 个整数寄存器，每个寄存器有 32 位，每位可以用一个带有写使能端的 D 触发器 dffe (图 2.23) 实现 (寄存器 0 除外，它永远是 0)。

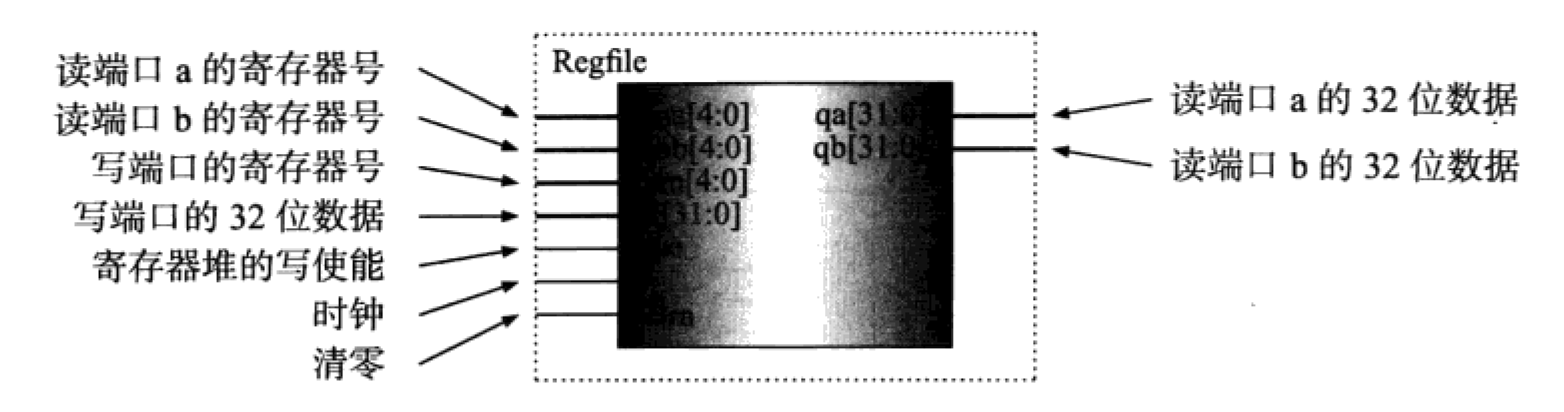


图 5.11 寄存器堆的电路符号及各信号的意义

图 5.12 示出的是由 32 个 dffe 组成一个 32 位寄存器 (dffe32) 的电路。

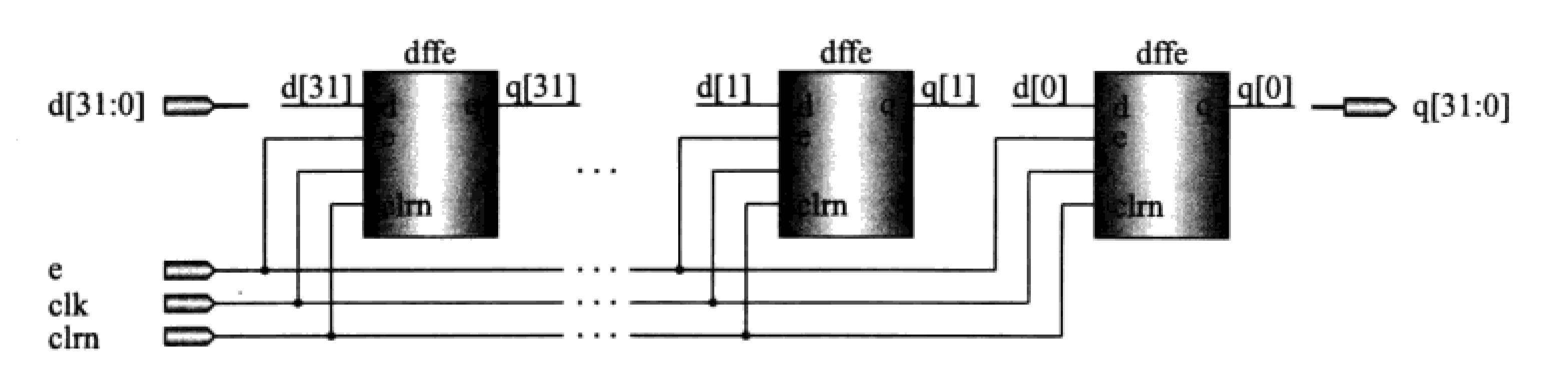


图 5.12 由 32 个 dffe 组成一个 32 位的寄存器 dffe32 的电路

图 5.13 示出的是由 31 个 dffe32 组成 32 个寄存器 (reg32) 的电路。由于 MIPS 规定寄存器 r0 的内容永远为 0，我们使用了 gnd 来实现它。注意输出信号 q[31:0][31:0] 的写法：第一个 [31:0] 表示有 32 个寄存器，第二个 [31:0] 表示每个寄存器有 32 位。虽然输入信号 d[31:0] 接到了每个寄存器的 d[31:0] 输入端，但每个寄存器都有各自的写使能端。被选中的寄存器的写使能端为 1，其余全部为 0。

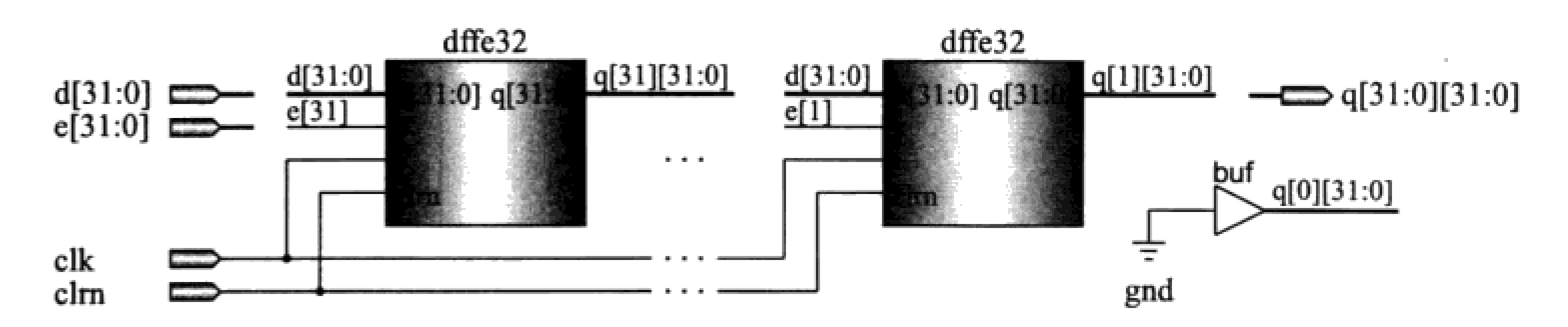


图 5.13 32 个 32 位的寄存器 reg32 的电路

有了 reg32，参照第 2 章，再设计一个带有使能端的 5-32 译码器 dec5e 和一个 32 位的 32 选 1 的多路器 mux32x32，按图 5.14 连接各个部件，我们就可以胜利完成寄存器堆的电路设计任务了。寄存器堆的两个读端口由两个 mux32x32 实现，它们都可以从 32 个寄存器中选出任意一个，把其内容送出。在写端口方面，译码器的 32 个输出信号 (写使能) 中最多有一个信号输出为 1，其余全部为 0。在时钟上升沿处，只把 d[31:0] 的数据写入写使能端为 1 的寄存器中。当译码器的使能端 ena 输入为 0 时，译码器的 32 个输出信号全部为 0，禁止向任何寄存器写入。

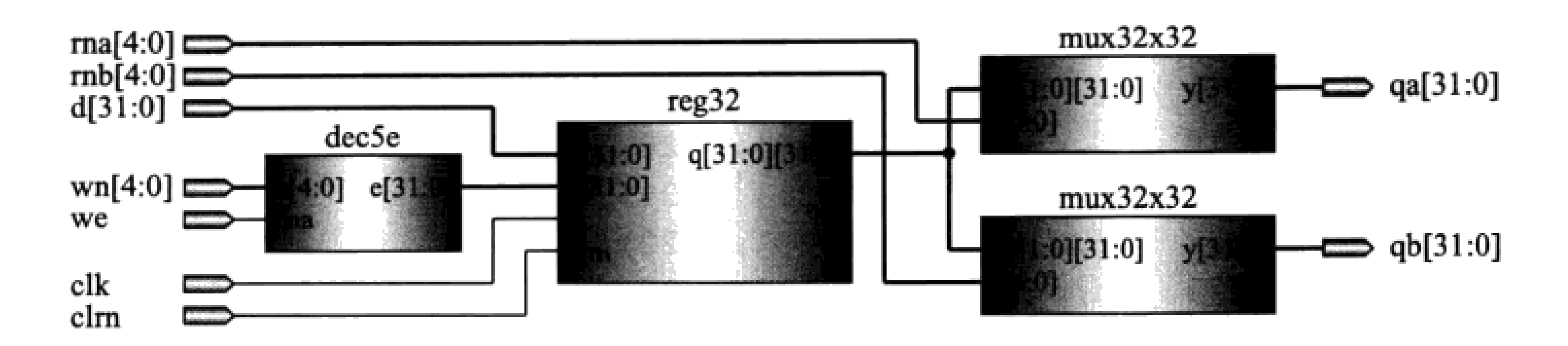


图 5.14 寄存器堆 regfile 的电路

5.2.2 结构描述风格的寄存器堆 Verilog HDL 代码

以下是寄存器堆的最笨的 Verilog HDL 代码 (应该说是最朴素的代码)，与我们在 5.2.1 小节描述的设计方法基本类似。它调用 5-32 译码器 dec5e 和 32 位寄存器 dffe32 两个模块。输出端的多路选择器在代码中以 function 形式直接给出。

```
module regfile_dataflow (rna,rnb,d,wn,we,clk,clrn,qa,qb);
    input  [4:0] rna,rnb,wn;
    input  [31:0] d;
    input         we,clk,clrn;
    output  [31:0] qa,qb;
    wire  [31:0]   e;
    wire  [31:0]   r00,r01,r02,r03,r04,r05,r06,r07;
    wire  [31:0]   r08,r09,r10,r11,r12,r13,r14,r15;
    wire  [31:0]   r16,r17,r18,r19,r20,r21,r22,r23;
    wire  [31:0]   r24,r25,r26,r27,r28,r29,r30,r31;
    dec5e decoder  (wn,we,e);
    assign         r00 = 0;
    dffe32 reg01   (d,clk,clrn,e[01], r01);
    dffe32 reg02   (d,clk,clrn,e[02], r02);
    dffe32 reg03   (d,clk,clrn,e[03], r03);
    dffe32 reg04   (d,clk,clrn,e[04], r04);
    dffe32 reg05   (d,clk,clrn,e[05], r05);
    dffe32 reg06   (d,clk,clrn,e[06], r06);
    dffe32 reg07   (d,clk,clrn,e[07], r07);
    dffe32 reg08   (d,clk,clrn,e[08], r08);
    dffe32 reg09   (d,clk,clrn,e[09], r09);
    dffe32 reg10   (d,clk,clrn,e[10], r10);
    dffe32 reg11   (d,clk,clrn,e[11], r11);
    dffe32 reg12   (d,clk,clrn,e[12], r12);
    dffe32 reg13   (d,clk,clrn,e[13], r13);
    dffe32 reg14   (d,clk,clrn,e[14], r14);
    dffe32 reg15   (d,clk,clrn,e[15], r15);
    dffe32 reg16   (d,clk,clrn,e[16], r16);
    dffe32 reg17   (d,clk,clrn,e[17], r17);
```

```
dffe32 reg18  (d,clk,clrn,e[18], r18);
dffe32 reg19  (d,clk,clrn,e[19], r19);
dffe32 reg20  (d,clk,clrn,e[20], r20);
dffe32 reg21  (d,clk,clrn,e[21], r21);
dffe32 reg22  (d,clk,clrn,e[22], r22);
dffe32 reg23  (d,clk,clrn,e[23], r23);
dffe32 reg24  (d,clk,clrn,e[24], r24);
dffe32 reg25  (d,clk,clrn,e[25], r25);
dffe32 reg26  (d,clk,clrn,e[26], r26);
dffe32 reg27  (d,clk,clrn,e[27], r27);
dffe32 reg28  (d,clk,clrn,e[28], r28);
dffe32 reg29  (d,clk,clrn,e[29], r29);
dffe32 reg30  (d,clk,clrn,e[30], r30);
dffe32 reg31  (d,clk,clrn,e[31], r31);
assign        qa = select(r00,r01,r02,r03,r04,r05,r06,r07,
                          r08,r09,r10,r11,r12,r13,r14,r15,
                          r16,r17,r18,r19,r20,r21,r22,r23,
                          r24,r25,r26,r27,r28,r29,r30,r31,rna);
assign        qb = select(r00,r01,r02,r03,r04,r05,r06,r07,
                          r08,r09,r10,r11,r12,r13,r14,r15,
                          r16,r17,r18,r19,r20,r21,r22,r23,
                          r24,r25,r26,r27,r28,r29,r30,r31,rnb);
function [31:0] select;
    input [31:0] r00,r01,r02,r03,r04,r05,r06,r07,
                 r08,r09,r10,r11,r12,r13,r14,r15,
                 r16,r17,r18,r19,r20,r21,r22,r23,
                 r24,r25,r26,r27,r28,r29,r30,r31;
    input [4:0]  s;
    case (s)
        5'd00: select = r00;
        5'd01: select = r01;
        5'd02: select = r02;
        5'd03: select = r03;
        5'd04: select = r04;
        5'd05: select = r05;
        5'd06: select = r06;
        5'd07: select = r07;
        5'd08: select = r08;
        5'd09: select = r09;
        5'd10: select = r10;
        5'd11: select = r11;
        5'd12: select = r12;
        5'd13: select = r13;
        5'd14: select = r14;
        5'd15: select = r15;
```

```
            5'd16: select = r16;
            5'd17: select = r17;
            5'd18: select = r18;
            5'd19: select = r19;
            5'd20: select = r20;
            5'd21: select = r21;
            5'd22: select = r22;
            5'd23: select = r23;
            5'd24: select = r24;
            5'd25: select = r25;
            5'd26: select = r26;
            5'd27: select = r27;
            5'd28: select = r28;
            5'd29: select = r29;
            5'd30: select = r30;
            5'd31: select = r31;
        endcase
    endfunction
endmodule
```

以下是 5-32 译码器的 Verilog HDL 代码。

```
module dec5e (n,ena,e);
    input [4:0] n;
    input       ena;
    output [31:0] e;
    function [31:0] decoder;
        input [4:0] n;
        input       ena;
        if (ena==1'b0) decoder = 32'h00000000;
        else begin
            case (n)
                5'b00000: decoder = 32'h00000001;
                5'b00001: decoder = 32'h00000002;
                5'b00010: decoder = 32'h00000004;
                5'b00011: decoder = 32'h00000008;
                5'b00100: decoder = 32'h00000010;
                5'b00101: decoder = 32'h00000020;
                5'b00110: decoder = 32'h00000040;
                5'b00111: decoder = 32'h00000080;
                5'b01000: decoder = 32'h00000100;
                5'b01001: decoder = 32'h00000200;
                5'b01010: decoder = 32'h00000400;
                5'b01011: decoder = 32'h00000800;
                5'b01100: decoder = 32'h00001000;
                5'b01101: decoder = 32'h00002000;
                5'b01110: decoder = 32'h00004000;
```

```
                5'b01111: decoder = 32'h00008000;
                5'b10000: decoder = 32'h00010000;
                5'b10001: decoder = 32'h00020000;
                5'b10010: decoder = 32'h00040000;
                5'b10011: decoder = 32'h00080000;
                5'b10100: decoder = 32'h00100000;
                5'b10101: decoder = 32'h00200000;
                5'b10110: decoder = 32'h00400000;
                5'b10111: decoder = 32'h00800000;
                5'b11000: decoder = 32'h01000000;
                5'b11001: decoder = 32'h02000000;
                5'b11010: decoder = 32'h04000000;
                5'b11011: decoder = 32'h08000000;
                5'b11100: decoder = 32'h10000000;
                5'b11101: decoder = 32'h20000000;
                5'b11110: decoder = 32'h40000000;
                5'b11111: decoder = 32'h80000000;
            endcase
        end
    endfunction
    assign e = decoder(n,ena);
endmodule
```

以下是 32 位寄存器的 Verilog HDL 代码。

```
module dffe32 (d,clk,clrn,e,q);
    input [31:0] d;
    input        clk,clrn,e;
    output [31:0] q;
    reg [31:0]    q;
    always @(negedge clrn or posedge clk)
        if (clrn == 0) begin
            q <= 0;
        end else begin
            if (e) q <= d;
        end
endmodule
```

5.2.3 功能描述风格的寄存器堆 Verilog HDL 代码

数据流描述风格的代码太麻烦了。以下的 Verilog HDL 代码同样实现寄存器堆电路。关键部分是类似于二维数组的寄存器变量：reg [31:0] register [1:31]，其中的 [31:0] 声明每个寄存器有 32 位，[1:31] 声明有 31 个寄存器 (寄存器 0 的内容永远是 0)。另外，我们使用了 integer 类型，它声明变量 i 是整数类型。整数类型的变量是 32 位的带符号数。与 reg 和 wire 不同，我们不能对 integer 变量做“按位”的操作。实际上，integer 变量本身不会以逻辑电路的形式出现在经过逻辑综合后的电路中。

```
module regfile (rna,rnb,d,wn,we,clk,clrn,qa,qb);
    input   [4:0] rna,rnb,wn;
    input  [31:0] d;
    input         we,clk,clrn;
    output [31:0] qa,qb;
    reg    [31:0] register [1:31]; // 31 x 32-bit regs

    // 2 read ports
    assign qa = (rna == 0) ? 0 : register[rna];
    assign qb = (rnb == 0) ? 0 : register[rnb];

    // 1 write port
    always @(posedge clk or negedge clrn)
        if (clrn == 0) begin
            integer i;
            for (i=1; i<32; i=i+1)
                register[i] <= 0;
        end else if ((wn != 0) && we)
            register[wn] <= d;
endmodule
```

比起数据流描述风格，功能描述风格的寄存器堆的 Verilog HDL 代码简洁多了。

5.3 数据路径设计

CPU 的电路包括数据路径 (Datapath) 和控制部件 (Control Unit) 两大部分。本节介绍单周期 CPU 的总体数据路径，并给出 Verilog HDL 代码。

5.3.1 多路选择器的使用

我们在 5.1 节已经分别介绍了每条指令执行时所需的硬件电路。有些部件的输入源不止一个，比如 ALU 的 b 输入端：当 CPU 执行诸如 add rd, rs, rt 的指令时，由 rt 指定的寄存器的内容应送到 ALU 的 b 输入端；而当 CPU 执行诸如 addi rt, rs, imm 的指令时，立即数 imm 经符号扩展后应送到 ALU 的 b 输入端。解决这种输入源冲突问题的办法是使用多路选择器。

1. 下一条指令地址的选择 (控制信号 pcsource)

程序不转移时下一条指令的地址是 PC + 4，但以下五条指令会引起程序转移。

```
beq/bne rs,rt,label # if ... pc <-- label   | op | rs | rt |    offset     |
jr      rs          # pc <-- rs             | op | rs | 0  |  0  | 0 |func|
j/jal   address     # pc <-- address*4      | op |          address          |
```

beq 指令从寄存器堆的 rs 寄存器和 rt 寄存器读出两个 32 位数据，将它们分别送至 ALU 的 a 和 b 输入端。ALU 比较这两个数是否相等。若相等，ALU 的输出 z 为 1，程序发生转移。转移的目标地址为 PC 加 4，再加上符号扩展的偏移量左移两位。bne 是不相等时转移。jr 指令比较简单，rs 寄存器中的内容就是转移目标地址。j 和 jal 指令中的 26 位地址左移两位，再与 PC + 4 的高 4 位拼接在一起，即是转移目标地址。

算上不转移时的 PC + 4，下一条指令地址有 4 个来源。我们使用一个四选一多路器来选择下一条指令的地址，如图 5.15 所示。

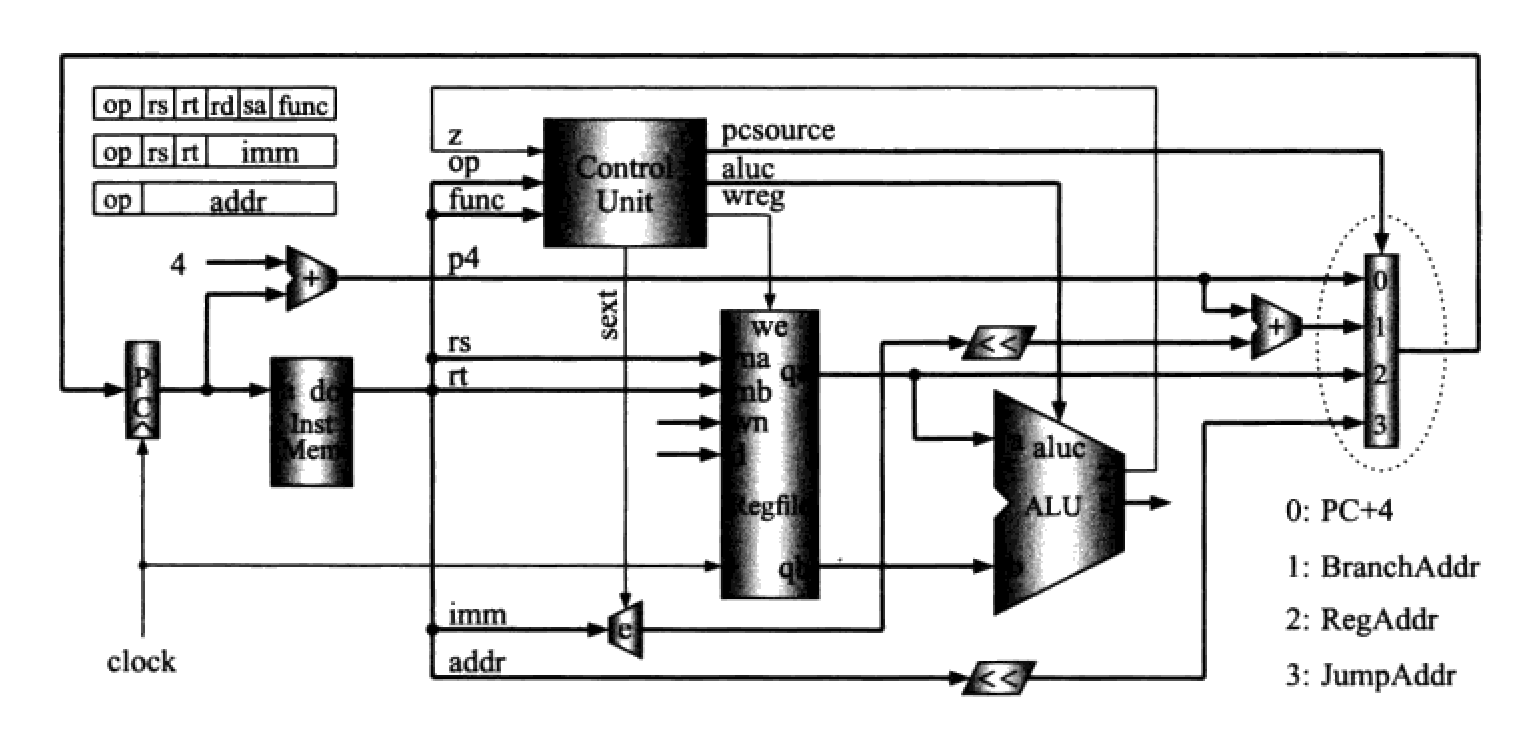


图 5.15 下一条指令地址的选择 (四个数据源)

2. ALU 的 a 输入端 (控制信号 shift)

考虑以下两条寄存器操作型指令。一条是加法指令，一条是移位指令。

```
add rd,rs,rt ; rd <-- rs + rt
sll rd,rt,sa ; rd <-- rt << sa
```

op	rs	rt	rd	0	func
op	0	rt	rd	sa	func

add 指令从寄存器堆的 rs 寄存器和 rt 寄存器读出两个 32 位数据，将它们分别送至 ALU 的 a 和 b 输入端。ALU 的加法结果保存到寄存器堆的 rd 寄存器中。sll 指令从寄存器堆的 rt 寄存器读出一个 32 位数据，送它至 ALU b 输入端。指令中的 sa (Shift Amount) 送至 ALU a 输入端的低 5 位，高 27 位随便填些什么，比如填 0。

因此，ALU 的 a 输入端有两个数据源：一个来自寄存器堆的 qa 端，一个来自指令中的 sa。我们使用一个二选一多路器从两个数据源中选出一个送至 ALU 的 a 输入端，见图 5.16。多路器的选择信号命名为 shift，其意义为 shift = 1 时选择 sa。

3. ALU 的 b 输入端和寄存器堆的 wn 输入端 (控制信号 aluimm 和 regrt)

考虑以下两条指令。一条是寄存器型指令，一条是立即数型指令。

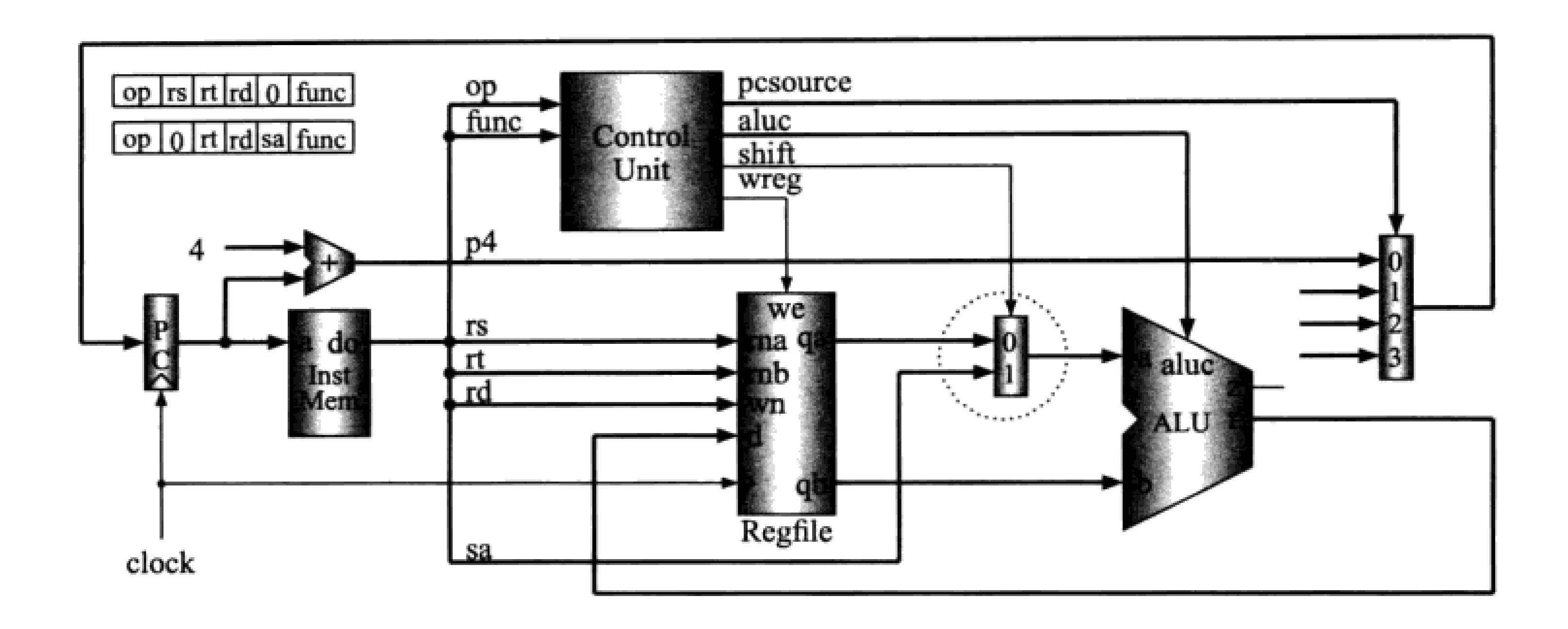


图 5.16 ALU 输入端 a 的两个数据源

```
add  rd,rs,rt  ; rd <-- rs + rt
addi rt,rs,imm ; rt <-- rs + imm
```

op	rs	rt	rd	0	func
op	rs	rt	imm		

add 指令从寄存器堆的 rt 寄存器读出 32 位数据将它送至 ALU 的 b 输入端。ALU 的加法结果保存到寄存器堆的 rd 寄存器中。addi 指令把指令中的立即数经符号扩展后送至 ALU 的 b 输入端。ALU 的加法结果保存到 rt 寄存器中。

因此，ALU 的 b 输入端有两个数据源：一个来自寄存器堆的 qb 端，一个来自指令中的 imm。寄存器堆的 wn 输入端 (目的寄存器号) 也有两个数据源：一个来自指令中的 rd，一个来自指令中的 rt。我们使用两个二选一多路器，如图 5.17 所示。两个二选一多路器的选择信号分别为 aluimm 和 regrt。当 aluimm = 1 时，选择立即数，否则，选择寄存器操作数；当 regrt = 1 时，选择 rt，否则，选择 rd。

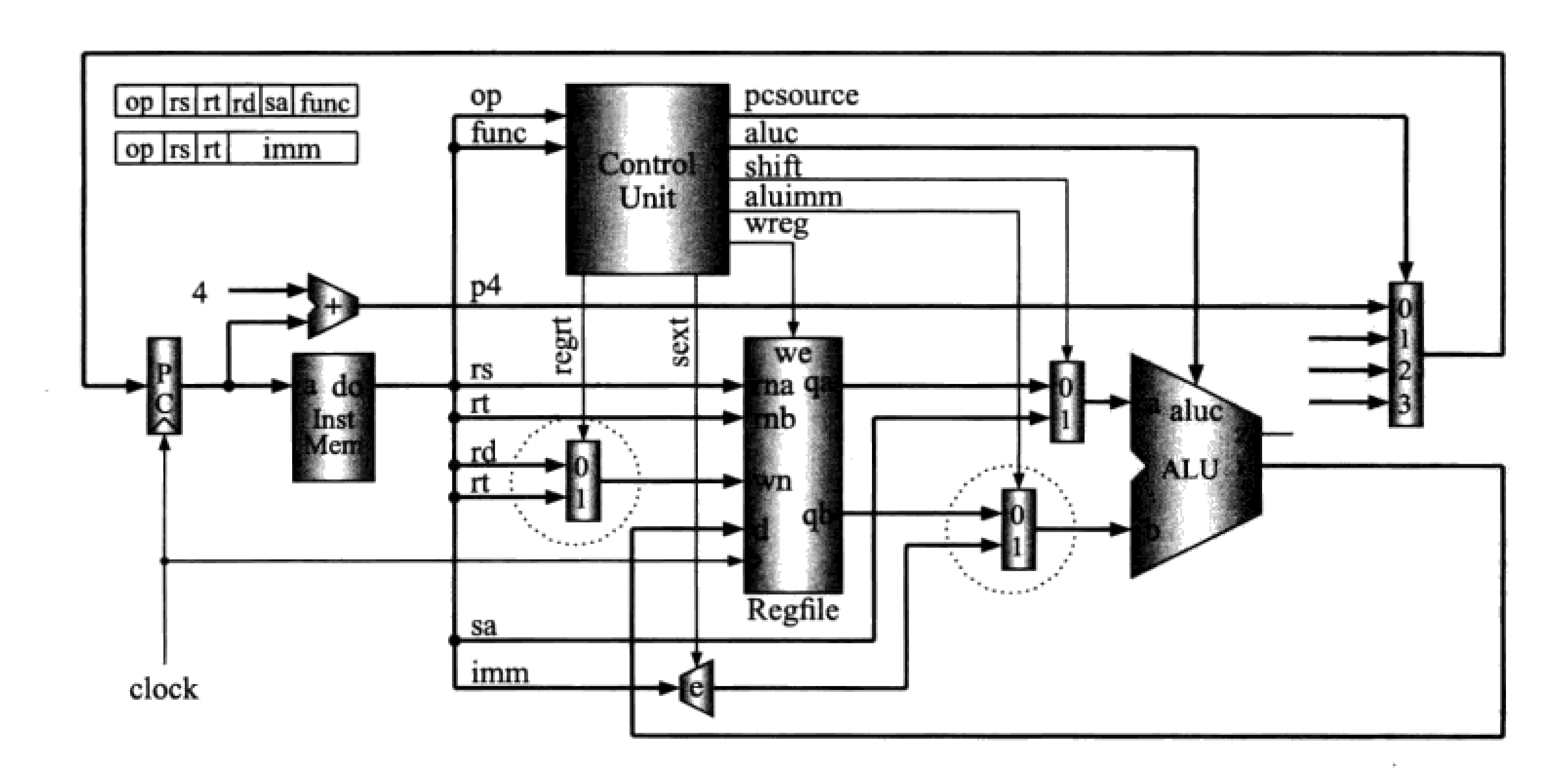


图 5.17 ALU 输入端 b 的两个数据源和寄存器堆输入端 wn 的两个数据源

4. 寄存器堆的 d 输入端 (控制信号 m2reg 和 jal)

考虑以下三条指令：寄存器型指令、存储器访问指令和子程序调用指令。

```
add rd,rs,rt       ; rd <-- rs + rt                 | op | rs | rt | rd | 0 | func |
lw  rt,offset(rs)  ; rt <-- mem[rs+offset]          | op | rs | rt |     offset    |
jal address        ; r31<--pc+4, pc<--address*4     | op |       address           |
```

add 指令把 ALU 结果写入寄存器堆；lw 指令把从存储器取来的数据写入寄存器堆；jal 指令把 PC + 4 写入寄存器堆，见图 5.18。

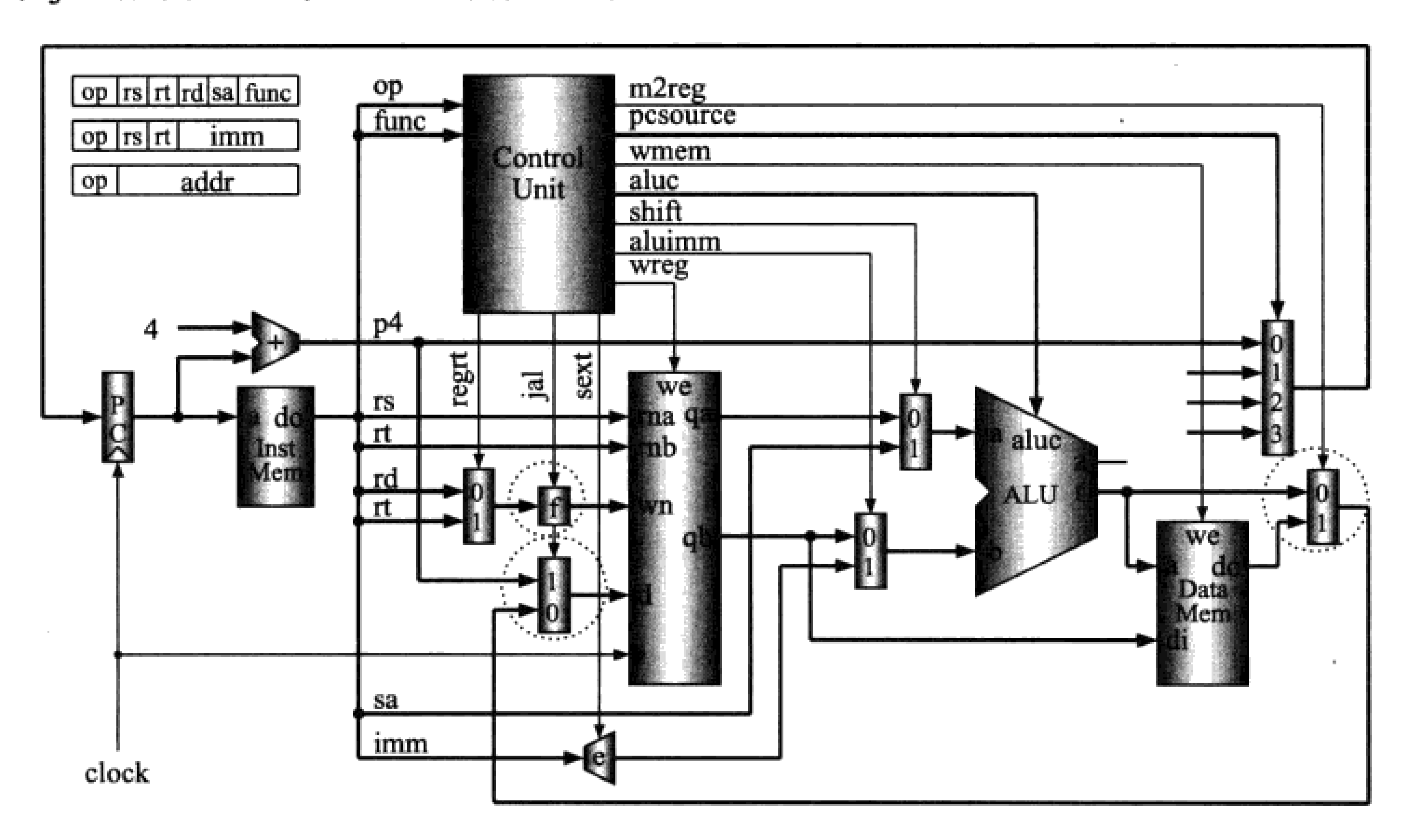


图 5.18 寄存器堆输入端 d 的三个数据源

我们使用了两个二选一多路器从三个数据源中选出一个。两个多路器的选择信号分别为 m2reg 和 jal。由于 jal 指令总是把返回地址写入寄存器 r31 中，因此我们在电路中增加了一个小小模块，即图中的 f。模块 f 的输入为 5 位的寄存器号 reg_dest 和控制信号 jal，输出为 wn，Verilog HDL 语句为：

```
assign wn = reg_dest | {5{jal}};
```

5.3.2 单周期 CPU 的总体电路

综合以上讨论的各种情况，我们得到如图 5.19 所示的单周期 CPU 的总体电路。该电路必须能够执行表 4.4 列出的所有 20 条指令。

从概念上讲，CPU 部分并不包括存储器 (或主存)。如果把指令存储器和数据存储器抽出来，并把 CPU 部分用一个器件符号表示，则有图 5.20 所示的结构。我们最好称其为单周期计算机，因为计算机包括了 CPU 和存储器。

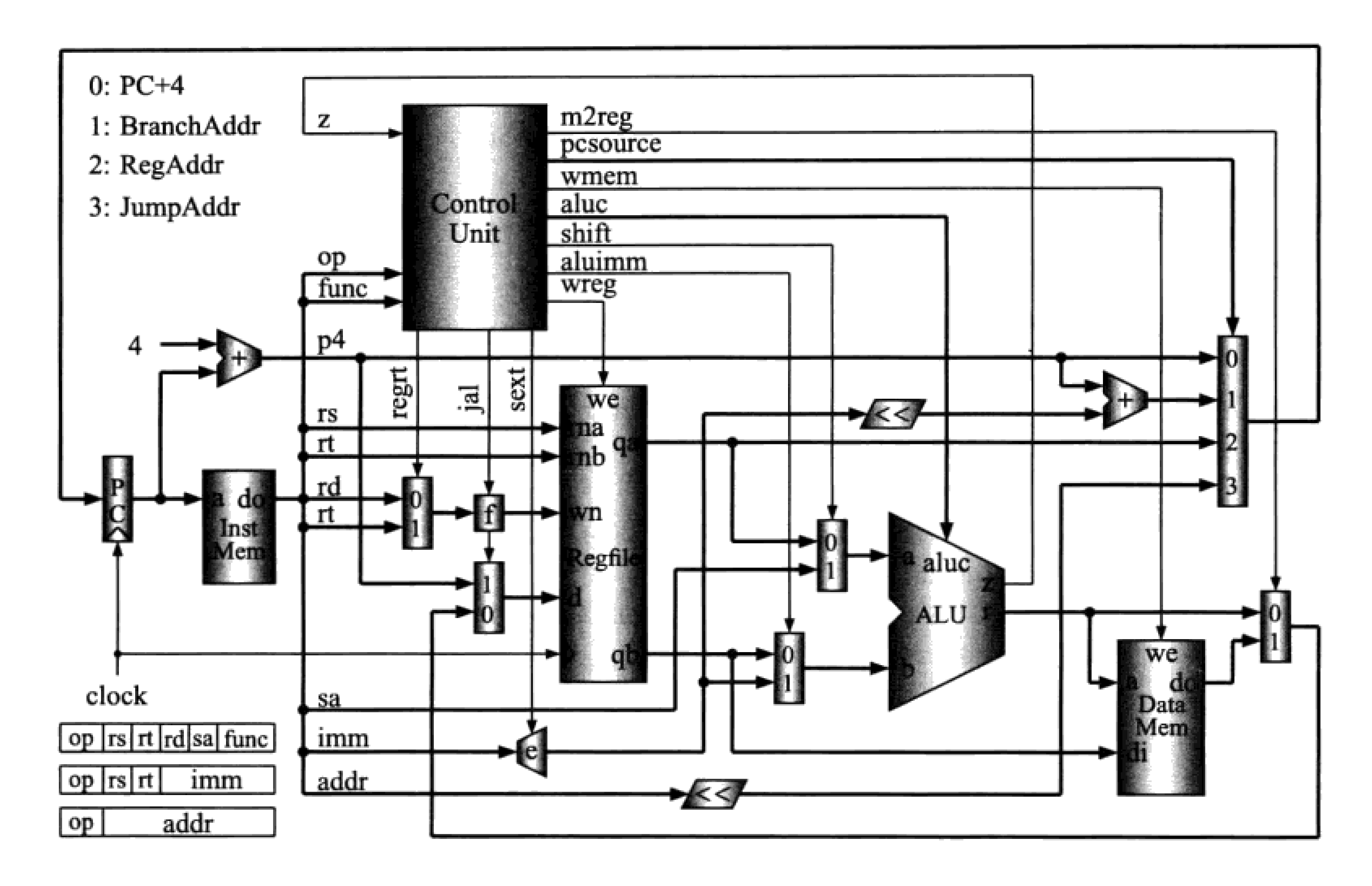


图 5.19 单周期 CPU + 指令存储器 + 数据存储器的总体电路图

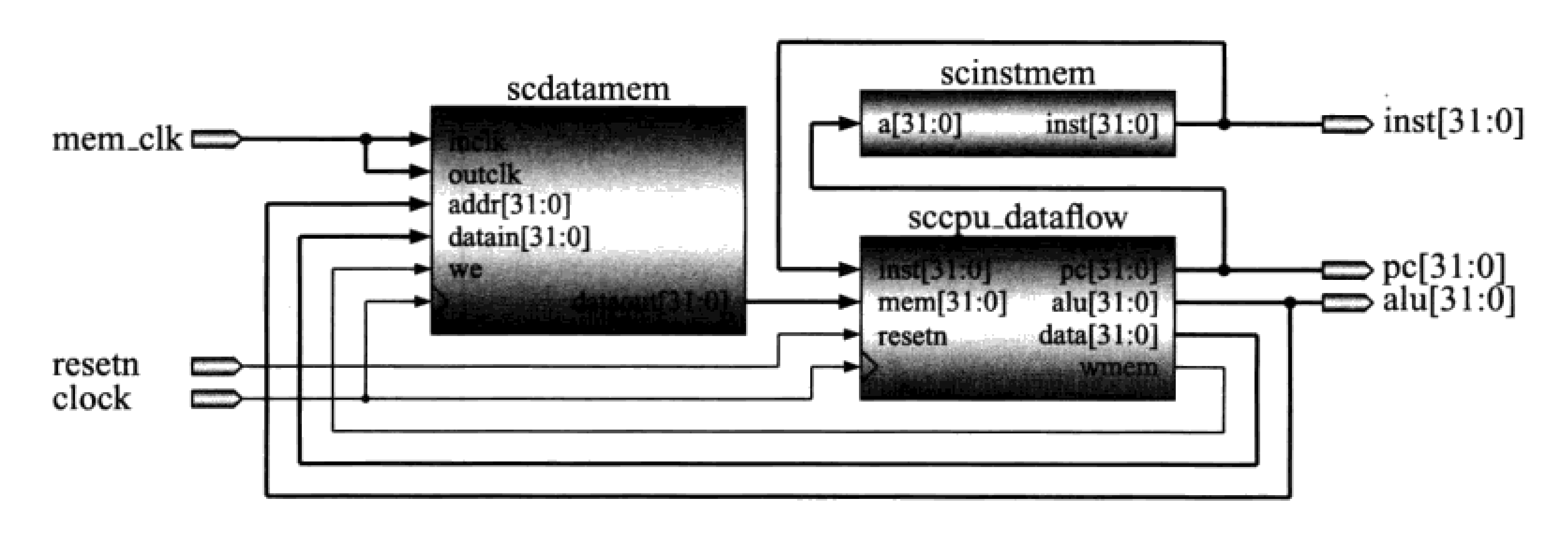


图 5.20 单周期 CPU + 指令存储器 + 数据存储器的模块图

5.3.3 单周期 CPU 的 Verilog HDL 代码

以下是图 5.20 所示的单周期计算机的 Verilog HDL 代码。它调用单周期 CPU 模块 sccpu_dataflow、指令存储器模块 scinstmem 和数据存储器模块 scdatamem。将在稍后讨论控制部件和存储器模块的设计方法。

```
module sccomp_dataflow (clock,resetn,inst,pc,aluout,memout,mem_clk);
    input clock,resetn,mem_clk;
    output [31:0] inst,pc,aluout,memout;
```

```
    wire [31:0]   data;
    wire          wmem;
    sccpu_dataflow s (clock,resetn,inst,memout,pc,wmem,aluout,data);
    scinstmem imem (pc,inst);
    scdatamem dmem (clock,memout,data,aluout,wmem,mem_clk,mem_clk);
endmodule
```

以下是数据流描述风格的单周期 CPU 的 Verilog HDL 代码 sccpu_dataflow.v。

```
module sccpu_dataflow (clock,resetn,inst,mem,pc,wmem,alu,data);
    input  [31:0] inst,mem;
    input         clock,resetn;
    output [31:0] pc,alu,data;
    output        wmem;
    wire [31:0]   p4,bpc,npc,adr,ra,alua,alub,res,alu_mem;
    wire [3:0]    aluc;
    wire [4:0]    reg_dest,wn;
    wire [1:0]    pcsource;
    wire          zero,wmem,wreg,regrt,m2reg,shift,aluimm,jal,sext;
    wire [31:0]   sa = {27'b0,inst[10:6]};
    wire [31:0]   offset = {imm[13:0],inst[15:0],2'b00};
    sccu_dataflow cu (inst[31:26],inst[5:0],zero,wmem,wreg,regrt,
                        m2reg,aluc,shift,aluimm,pcsource,jal,sext);
    wire          e = sext & inst[15];
    wire [15:0]   imm = {16{e}};
    wire [31:0]   immediate = {imm,inst[15:0]};
    dff32 ip (npc,clock,resetn,pc);
    cla32 pcplus4 (pc,32'h4, 1'b0,p4);
    cla32 br_adr  (p4,offset,1'b0,adr);
    wire [31:0]   jpc = {p4[31:28],inst[25:0],2'b00};
    mux2x32 alu_b (data,immediate,aluimm,alub);
    mux2x32 alu_a (ra,sa,shift,alua);
    mux2x32 result (alu,mem,m2reg,alu_mem);
    mux2x32 link  (alu_mem,p4,jal,res);
    mux2x5 reg_wn (inst[15:11],inst[20:16],regrt,reg_dest);
    assign wn = reg_dest | {5{jal}}; // jal: r31 <-- p4;
    mux4x32 nextpc (p4,adr,ra,jpc,pcsource,npc);
    regfile rf (inst[25:21],inst[20:16],res,wn,wreg,clock,resetn,
                ra,data);
    alu al_unit (alua,alub,aluc,alu,zero);
endmodule
```

5.4 控制部件设计

本节详细描述单周期 CPU 中的控制部件的设计方法并给出 Verilog HDL 代码。

5.4.1 控制部件的逻辑设计

首先我们根据指令中的 6 位 op 对指令译码。如果 op = 0，则需要再检查 6 位的 func，见表 5.1。我们得到每条指令译码的逻辑表达式 (未化简)。

表 5.1 指令译码

指令	op[5:0]	func[5:0]	指令	op[5:0]
add	000000	100000	addi	001000
sub	000000	100010	andi	001100
and	000000	100100	ori	001101
or	000000	100101	xori	001110
xor	000000	100110	lw	100011
sll	000000	000000	sw	101011
srl	000000	000010	beq	000100
sra	000000	000011	bne	000101
jr	000000	001000	lui	001111
			j	000010
			jal	000011

$$
\begin{aligned}
\mathrm{r_type} &= \overline{op[5]}\ \overline{op[4]}\ \overline{op[3]}\ \overline{op[2]}\ \overline{op[1]}\ \overline{op[0]};\\
\mathrm{i_add} &= \mathrm{r_type}\ func[5]\ \overline{func[4]}\ \overline{func[3]}\ \overline{func[2]}\ \overline{func[1]}\ \overline{func[0]};\\
\mathrm{i_sub} &= \mathrm{r_type}\ func[5]\ \overline{func[4]}\ \overline{func[3]}\ \overline{func[2]}\ func[1]\ \overline{func[0]};\\
\mathrm{i_and} &= \mathrm{r_type}\ func[5]\ \overline{func[4]}\ \overline{func[3]}\ func[2]\ \overline{func[1]}\ \overline{func[0]};\\
\mathrm{i_or} &= \mathrm{r_type}\ func[5]\ \overline{func[4]}\ \overline{func[3]}\ func[2]\ \overline{func[1]}\ func[0];\\
\mathrm{i_xor} &= \mathrm{r_type}\ func[5]\ \overline{func[4]}\ \overline{func[3]}\ func[2]\ func[1]\ \overline{func[0]};\\
\mathrm{i_sll} &= \mathrm{r_type}\ \overline{func[5]}\ \overline{func[4]}\ \overline{func[3]}\ \overline{func[2]}\ \overline{func[1]}\ \overline{func[0]};\\
\mathrm{i_srl} &= \mathrm{r_type}\ \overline{func[5]}\ \overline{func[4]}\ \overline{func[3]}\ \overline{func[2]}\ func[1]\ \overline{func[0]};\\
\mathrm{i_sra} &= \mathrm{r_type}\ \overline{func[5]}\ \overline{func[4]}\ \overline{func[3]}\ \overline{func[2]}\ func[1]\ func[0];\\
\mathrm{i_jr} &= \mathrm{r_type}\ \overline{func[5]}\ \overline{func[4]}\ func[3]\ \overline{func[2]}\ \overline{func[1]}\ \overline{func[0]};\\
\mathrm{i_addi} &= \overline{op[5]}\ \overline{op[4]}\ op[3]\ \overline{op[2]}\ \overline{op[1]}\ \overline{op[0]};\\
\mathrm{i_andi} &= \overline{op[5]}\ \overline{op[4]}\ op[3]\ op[2]\ \overline{op[1]}\ \overline{op[0]};\\
\mathrm{i_ori} &= \overline{op[5]}\ \overline{op[4]}\ op[3]\ op[2]\ \overline{op[1]}\ op[0];\\
\mathrm{i_xori} &= \overline{op[5]}\ \overline{op[4]}\ op[3]\ op[2]\ op[1]\ \overline{op[0]};\\
\mathrm{i_lw} &= op[5]\ \overline{op[4]}\ \overline{op[3]}\ \overline{op[2]}\ op[1]\ op[0];\\
\mathrm{i_sw} &= op[5]\ \overline{op[4]}\ op[3]\ \overline{op[2]}\ op[1]\ op[0];\\
\mathrm{i_beq} &= \overline{op[5]}\ \overline{op[4]}\ \overline{op[3]}\ op[2]\ \overline{op[1]}\ \overline{op[0]};\\
\mathrm{i_bne} &= \overline{op[5]}\ \overline{op[4]}\ \overline{op[3]}\ op[2]\ \overline{op[1]}\ op[0];\\
\mathrm{i_lui} &= \overline{op[5]}\ \overline{op[4]}\ op[3]\ op[2]\ op[1]\ op[0];\\
\mathrm{i_j} &= \overline{op[5]}\ \overline{op[4]}\ \overline{op[3]}\ \overline{op[2]}\ op[1]\ \overline{op[0]};\\
\mathrm{i_jal} &= \overline{op[5]}\ \overline{op[4]}\ \overline{op[3]}\ \overline{op[2]}\ op[1]\ op[0];
\end{aligned}
$$

我们在每条指令的名称前面加了“i_”，这是为了避免在写 Verilog HDL 代码时与 Verilog HDL 关键字冲突。因为所有寄存器类型指令的 op 均相同，所以我们使用了一个临时变量 r_type。译出了指令之后，我们开始做重要的工作：确认每条指令执行时，控制信号应产生什么值。在我们的 CPU 中，控制信号有 10 个，见表 5.2。

表 5.2　控制信号的意义

控制信号	意 义	
wreg	写寄存器：	为 1 时写寄存器；否则不写
regrt	目的寄存器号是 rt：	为 1 时选择 rt；否则选择 rd
jal	子程序调用：	为 1 时表示指令是 jal；否则不是
m2reg	存储器数据写入寄存器：	为 1 时选择存储器数据；否则选择 ALU 结果
shift	ALU a 使用移位位数：	为 1 时使用移位位数；否则使用寄存器数据
aluimm	ALU b 使用立即数：	为 1 时使用立即数；否则使用寄存器数据
sext	立即数符号扩展：	为 1 时符号扩展；否则零扩展
aluc[3:0]	ALU 操作控制：	见第 4 章图 4.6
wmem	写存储器：	为 1 时写存储器；否则不写
pcsource[1:0]	下一条指令地址的选择：	00 选 PC + 4；01 选转移地址； 10 选寄存器内的地址；11 选跳转地址

根据每条指令及控制信号的意义，我们可以写出控制信号的真值表，见表 5.3。我们从中选一条 lw 指令来说明填表的方法：lw 指令计算存储器地址时由 ALU 做加法 (aluc[3:0] = x000)；一个操作数来自寄存器堆 (shift = 0)；另一个操作数是指令中的偏移量 (aluimm = 1)；而且偏移量要符号扩展 (sext = 1)；要把结果写入寄存器堆 (wreg = 1)；而且是把存储器数据写入寄存器 (m2reg = 1)；目的寄存器号要选择 rt (regrt = 1)；不是 jal 指令 (jal = 0)；不写存储器 (wmem = 0)；下一条指令的地址是 PC + 4 (pcsource[1:0] = 00)。由表 5.3，我们可以写出各个控制信号的逻辑表达式：

```
wreg      = i_add + i_sub + i_and + i_or + i_xor + i_sll + i_srl + i_sra +
            i_addi + i_andi + i_ori + i_xori + i_lw + i_lui + i_jal;
regrt     = i_addi + i_andi + i_ori + i_xori + i_lw + i_lui;
jal       = i_jal;
m2reg     = i_lw;
shift     = i_sll + i_srl + i_sra;
aluimm    = i_addi + i_andi + i_ori + i_xori + i_lw + i_lui + i_sw;
sext      = i_addi + i_lw + i_sw + i_beq + i_bne;
aluc[3]   = i_sra;
aluc[2]   = i_sub + i_or + i_srl + i_sra + i_ori + i_lui;
aluc[1]   = i_xor + i_sll + i_srl + i_sra + i_xori + i_beq + i_bne + i_lui;
aluc[0]   = i_and + i_or + i_sll + i_srl + i_sra + i_andi + i_ori;
wmem      = i_sw;
```

表 5.3 控制信号的真值表

指令	z	wreg	regrt	jal	m2reg	shift	aluimm	sext	aluc[3:0]	wmem	pcsource[1:0]
add	x	1	0	0	0	0	0	x	x 0 0 0	0	0 0
sub	x	1	0	0	0	0	0	x	x 1 0 0	0	0 0
and	x	1	0	0	0	0	0	x	x 0 0 1	0	0 0
or	x	1	0	0	0	0	0	x	x 1 0 1	0	0 0
xor	x	1	0	0	0	0	0	x	x 0 1 0	0	0 0
sll	x	1	0	0	0	1	0	x	0 0 1 1	0	0 0
srl	x	1	0	0	0	1	0	x	0 1 1 1	0	0 0
sra	x	1	0	0	0	1	0	x	1 1 1 1	0	0 0
jr	x	0	x	x	x	x	x	x	x x x x	0	1 0
addi	x	1	1	0	0	0	1	1	x 0 0 0	0	0 0
andi	x	1	1	0	0	0	1	0	x 0 0 1	0	0 0
ori	x	1	1	0	0	0	1	0	x 1 0 1	0	0 0
xori	x	1	1	0	0	0	1	0	x 0 1 0	0	0 0
lw	x	1	1	0	1	0	1	1	x 0 0 0	0	0 0
sw	x	0	x	x	x	0	1	1	x 0 0 0	1	0 0
beq	0	0	x	x	x	0	0	1	x 0 1 0	0	0 0
beq	1	0	x	x	x	0	0	1	x 0 1 0	0	0 1
bne	0	0	x	x	x	0	0	1	x 0 1 0	0	0 1
bne	1	0	x	x	x	0	0	1	x 0 1 0	0	0 0
lui	x	1	1	0	0	x	1	x	x 1 1 0	0	0 0
j	x	0	x	x	x	x	x	x	x x x x	0	1 1
jal	x	1	x	1	x	x	x	x	x x x x	0	1 1

$$\text{pcsource}[1] = \text{i_jr} + \text{i_j} + \text{i_jal};$$
$$\text{pcsource}[0] = \text{i_beq}\ z + \text{i_bne}\ \overline{z} + \text{i_j} + \text{i_jal};$$

注意，如果一个信号名有多位 (例如 aluc)，分别写出每一位的逻辑表达式。使用以上表达式，我们可以用逻辑图输入的方法设计出单周期 CPU 的控制部件。具体的电路图我们就不给了，因为 5.4.2 小节的 Verilog HDL 代码与电路图完全等价。

单周期 CPU 控制部件是一个组合电路，所有输出信号大致可以分为 3 类：多路器的选择信号、写使能信号和 ALU 操作的控制信号。我们可以对逻辑表达式进行化简，但化简不是本节的重点，因此以上给出的表达式都没有经过化简。

5.4.2 控制部件的 Verilog HDL 代码

根据指令译码的逻辑表达式和控制信号的逻辑表达式，我们可以很快写出单周期 CPU 控制部件的 Verilog HDL 代码，如下。前半部分是指令译码，后半部分是控制信号产生。为了节省版面，删除了从 input 语句开始本应在左侧留有的 4 个空格。

```
module sccu_dataflow (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
                      aluimm, pcsource, jal, sext);
input [5:0] op,func;
input z;
output wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem;
output [3:0] aluc;
output [1:0] pcsource;

wire r_type = ~|op;
wire i_add  = r_type& func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
wire i_sub  = r_type& func[5]&~func[4]&~func[3]&~func[2]& func[1]&~func[0];
wire i_and  = r_type& func[5]&~func[4]&~func[3]& func[2]&~func[1]&~func[0];
wire i_or   = r_type& func[5]&~func[4]&~func[3]& func[2]&~func[1]& func[0];
wire i_xor  = r_type& func[5]&~func[4]&~func[3]& func[2]& func[1]&~func[0];
wire i_sll  = r_type&~func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
wire i_srl  = r_type&~func[5]&~func[4]&~func[3]&~func[2]& func[1]&~func[0];
wire i_sra  = r_type&~func[5]&~func[4]&~func[3]&~func[2]& func[1]& func[0];
wire i_jr   = r_type&~func[5]&~func[4]& func[3]&~func[2]&~func[1]&~func[0];
wire i_addi = ~op[5]&~op[4]& op[3]&~op[2]&~op[1]&~op[0];
wire i_andi = ~op[5]&~op[4]& op[3]& op[2]&~op[1]&~op[0];
wire i_ori  = ~op[5]&~op[4]& op[3]& op[2]&~op[1]& op[0];
wire i_xori = ~op[5]&~op[4]& op[3]& op[2]& op[1]&~op[0];
wire i_lw   =  op[5]&~op[4]&~op[3]&~op[2]& op[1]& op[0];
wire i_sw   =  op[5]&~op[4]& op[3]&~op[2]& op[1]& op[0];
wire i_beq  = ~op[5]&~op[4]&~op[3]& op[2]&~op[1]&~op[0];
wire i_bne  = ~op[5]&~op[4]&~op[3]& op[2]&~op[1]& op[0];
wire i_lui  = ~op[5]&~op[4]& op[3]& op[2]& op[1]& op[0];
wire i_j    = ~op[5]&~op[4]&~op[3]&~op[2]& op[1]&~op[0];
wire i_jal  = ~op[5]&~op[4]&~op[3]&~op[2]& op[1]& op[0];

assign wreg        = i_add |i_sub |i_and|i_or  |i_xor|i_sll|i_srl|i_sra|
                     i_addi|i_andi|i_ori|i_xori|i_lw |i_lui|i_jal;
assign regrt       = i_addi|i_andi|i_ori|i_xori|i_lw |i_lui;
assign jal         = i_jal;
assign m2reg       = i_lw;
assign shift       = i_sll|i_srl|i_sra;
assign aluimm      = i_addi|i_andi|i_ori|i_xori|i_lw|i_lui|i_sw;
assign sext        = i_addi|i_lw |i_sw|i_beq|i_bne;
assign aluc[3]     = i_sra;
assign aluc[2]     = i_sub|i_or|i_srl|i_sra|i_ori|i_lui;
assign aluc[1]     = i_xor|i_sll|i_srl|i_sra|i_xori|i_beq|i_bne|i_lui;
assign aluc[0]     = i_and|i_or|i_sll|i_srl|i_sra|i_andi|i_ori;
assign wmem        = i_sw;
assign pcsource[1] = i_jr|i_j|i_jal;
assign pcsource[0] = i_beq&z|i_bne&~z|i_j|i_jal;
endmodule
```

5.5 存储器及测试程序设计

CPU 已设计完毕。本节讨论两个存储器模块的设计并描述如何开发一个测试程序来验证我们所设计的 CPU 电路的正确性。

5.5.1 数据存储器设计

使用 Verilog HDL 可以声明存储器类型的变量，但我们这里使用 Altera 提供的 LPM (Library of Parameterized Modules) 中的 ram 模块 lpm_ram_dq 实现我们的数据存储器。本章最后给出一般的、不依赖任何特定器件的代码。

以下是它的 Verilog HDL 代码。我们使用时钟的后半个周期写存储器。由于 Quartus II 本身的限制，我们还使用了两个专门的时钟 (inclk 和 outclk) 用于访问同步存储器。理想的情况是不需要这两个时钟，但 Quartus II 不支持异步存储器。

```
module scdatamem (clk, dataout, datain, addr, we, inclk, outclk);
    input  [31:0]  datain;
    input  [31:0]  addr;
    input          clk, we, inclk, outclk;
    output [31:0]  dataout;
    wire           write_enable = we & ~clk;
    lpm_ram_dq ram (.data (datain),
                    .address (addr[6:2]),
                    .we (write_enable),
                    .inclock (inclk),
                    .outclock (outclk),
                    .q (dataout));
    defparam   ram.lpm_width   = 32;
    defparam   ram.lpm_widthad = 5;
    defparam   ram.lpm_indata  = "registered";
    defparam   ram.lpm_outdata = "registered";
    defparam   ram.lpm_file    = "scdatamem.mif";
    defparam   ram.lpm_address_control = "registered";
endmodule
```

注意此处模块参数的代入使用了与以往不同的方式，这种方式的优点是各信号的次序可以是任意的。代码中的 lpm_file 参数用于指定存储器的初始化文件。以下是存储器初始化文件 scdatamem.mif 的内容。我们设置了 4 个数据，测试程序要把它们全部加在一起。注意，mif 文件中不能有汉字。

```
DEPTH = 32;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
```

```
BEGIN
[0..1F] : 00000000; % Range--Every address from 0 to 1F = 0    %
  14 : 000000A3; % (50) data[0]    0 +  A3 =  A3               %
  15 : 00000027; % (54) data[1]   A3 +  27 =  CA               %
  16 : 00000079; % (58) data[2]   CA +  79 = 143               %
  17 : 00000115; % (5C) data[3]  143 + 115 = 258               %
END ;
```

最左一列是存储器的字地址，冒号右边的是相应存储单元的内容。一行中两个百分号之间的内容是注释。注释开始处括号中的十六进制数是字节地址。

5.5.2 指令存储器及测试程序设计

类似地，我们使用了 lpm_rom 实现指令存储器。我们要对 ROM 初始化。初始化文件由 lpm_file 参数指定。注意 Altera 的有些器件不支持异步 ROM。

```
module scinstmem (a,inst);
    input [31:0] a;
    output [31:0] inst;
    lpm_rom  lpm_rom_component (.address (a[6:2]),
                                .q (inst));
    defparam lpm_rom_component.lpm_width    = 32,
             lpm_rom_component.lpm_widthad  = 5,
             lpm_rom_component.lpm_numwords = "unused",
             lpm_rom_component.lpm_file     = "scinstmem.mif",
             lpm_rom_component.lpm_indata   = "unused",
             lpm_rom_component.lpm_outdata  = "unregistered",
             lpm_rom_component.lpm_address_control = "unregistered";
endmodule
```

指令存储器的初始化文件 (scinstmem.mif) 包含 CPU 的测试代码，如下。指令用十六进制表示，括号中的十六进制数是 PC 值。

```
DEPTH = 32;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
[0..1F] : 00000000; % Range--Every address from 0 to 1F = 00000000       %
 0 : 3c010000; % (00) main:  lui  r1, 0          # address of data[0]    %
 1 : 34240050; % (04)        ori  r4, r1, 80     # address of data[0]    %
 2 : 20050004; % (08)        addi r5, r0,  4     # counter               %
 3 : 0c000018; % (0c) call:  jal  sum            # call function         %
 4 : ac820000; % (10)        sw   r2, 0(r4)      # store result          %
 5 : 8c890000; % (14)        lw   r9, 0(r4)      # check sw              %
 6 : 01244022; % (18)        sub  r8, r9, r4     # sub: r8 <-- r9 - r4   %
```

```
 7 : 20050003; % (1c)        addi r5, r0,  3     # counter               %
 8 : 20a5ffff; % (20) loop2: addi r5, r5, -1     # counter - 1           %
 9 : 34a8ffff; % (24)        ori  r8, r5, 0xffff # zero-extend: 0000ffff %
 A : 39085555; % (28)        xori r8, r8, 0x5555 # zero-extend: 0000aaaa %
 B : 2009ffff; % (2c)        addi r9, r0, -1     # sign-extend: ffffffff %
 C : 312affff; % (30)        andi r10, r9,0xffff # zero-extend: 0000ffff %
 D : 01493025; % (34)        or   r6, r10, r9    # or:  ffffffff         %
 E : 01494026; % (38)        xor  r8, r10, r9    # xor: ffff0000         %
 F : 01463824; % (3c)        and  r7, r10, r6    # and: 0000ffff         %
10 : 10a00001; % (40)        beq  r5, r0, shift  # if r5 = 0, goto shift %
11 : 08000008; % (44)        j    loop2          # jump loop2            %
12 : 2005ffff; % (48) shift: addi r5, r0, -1     # r5   = ffffffff       %
13 : 000543c0; % (4c)        sll  r8, r5, 15     # <<15 = ffff8000       %
14 : 00084400; % (50)        sll  r8, r8, 16     # <<16 = 80000000       %
15 : 00084403; % (54)        sra  r8, r8, 16     # >>16 = ffff8000(arith) %
16 : 000843c2; % (58)        srl  r8, r8, 15     # >>15 = 0001ffff(logic) %
17 : 08000017; % (5c) finish: j   finish         # dead loop             %
18 : 00004020; % (60) sum:   add  r8, r0, r0     # sum                   %
19 : 8c890000; % (64) loop:  lw   r9, 0(r4)      # load data             %
1A : 20840004; % (68)        addi r4, r4,  4     # address + 4           %
1B : 01094020; % (6c)        add  r8, r8, r9     # sum                   %
1C : 20a5ffff; % (70)        addi r5, r5, -1     # counter - 1           %
1D : 14a0fffb; % (74)        bne  r5, r0, loop   # finish?               %
1E : 00081000; % (78)        sll  r2, r8, 0      # move result to v0     %
1F : 03e00008; % (7c)        jr   r31            # return                %
END ;
```

5.5.3 单周期 CPU 测试结果及说明

图 5.21 ~ 图 5.23 是在单周期 CPU 上运行测试程序 scinstmem.mif 时的波形图。开始处 resetn 的低电平对 CPU 进行复位操作，主要是把程序计数器和寄存器堆清零。CPU 从指令存储器的 0 地址开始取指令。每个时钟周期执行一条指令。所有的 20 条指令都出现在我们的测试程序中。注意 PC = 0000000C 处是一条 jal 指令：调用子程序 sum。0000007C 处返回。请读者检查每条指令的执行结果是否正确。

指令存储器可以不用 lpm_rom，直接使用以下的通用代码实现。注意，从功能上讲，ROM 只是一种组合电路。

```
module scinstmem (a,inst);
    input [31:0] a;
    output [31:0] inst;
    wire [31:0] rom [0:31];
    assign rom[5'h00] = 32'h3c010000; // (00) main:   lui  r1, 0
    assign rom[5'h01] = 32'h34240050; // (04)         ori  r4, r1, 80
    assign rom[5'h02] = 32'h20050004; // (08)         addi r5, r0,  4
    assign rom[5'h03] = 32'h0c000018; // (0c) call:   jal  sum
    assign rom[5'h04] = 32'hac820000; // (10)         sw   r2, 0(r4)
```

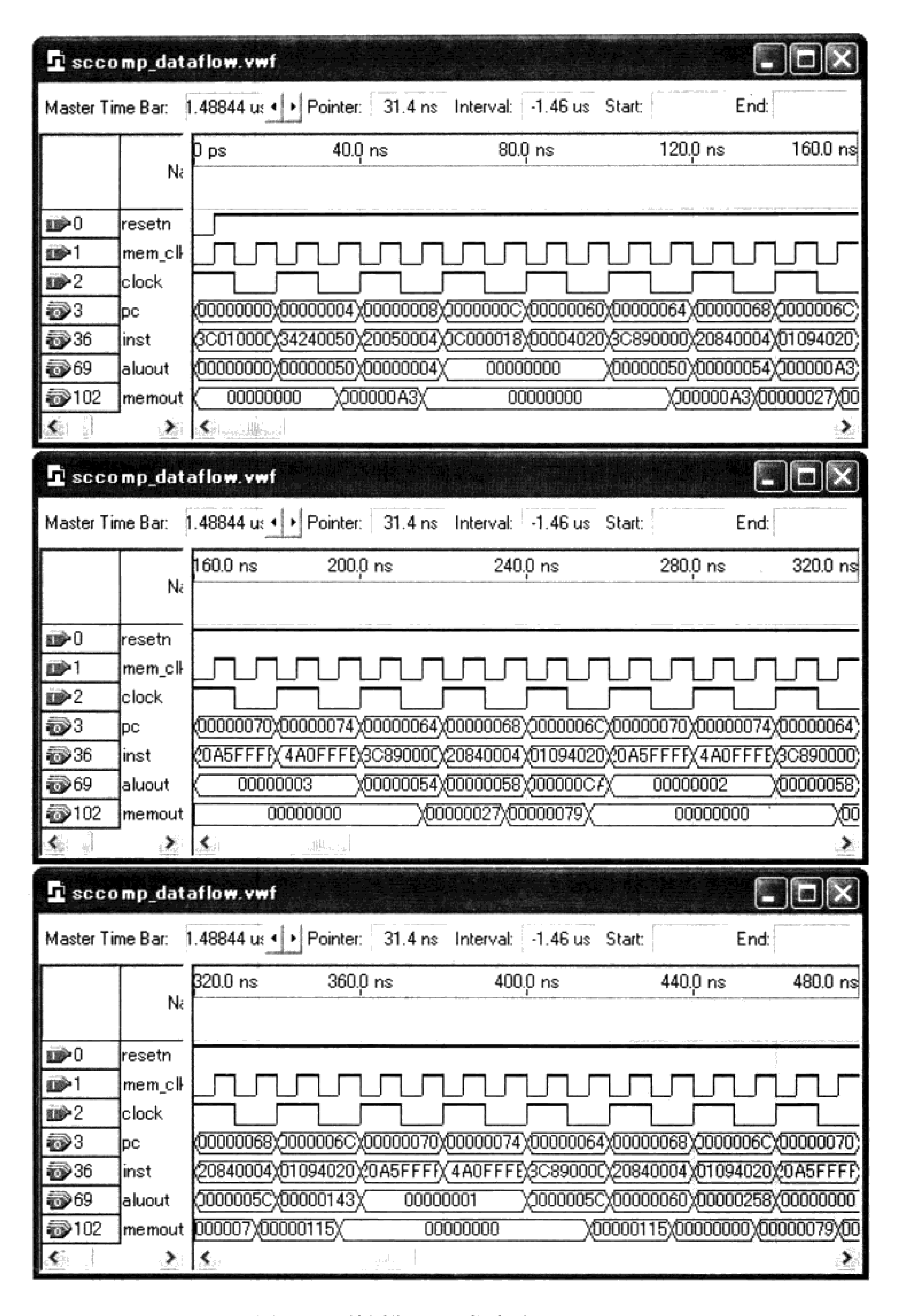


图 5.21　单周期 CPU 仿真波形图 (1 ～ 3)

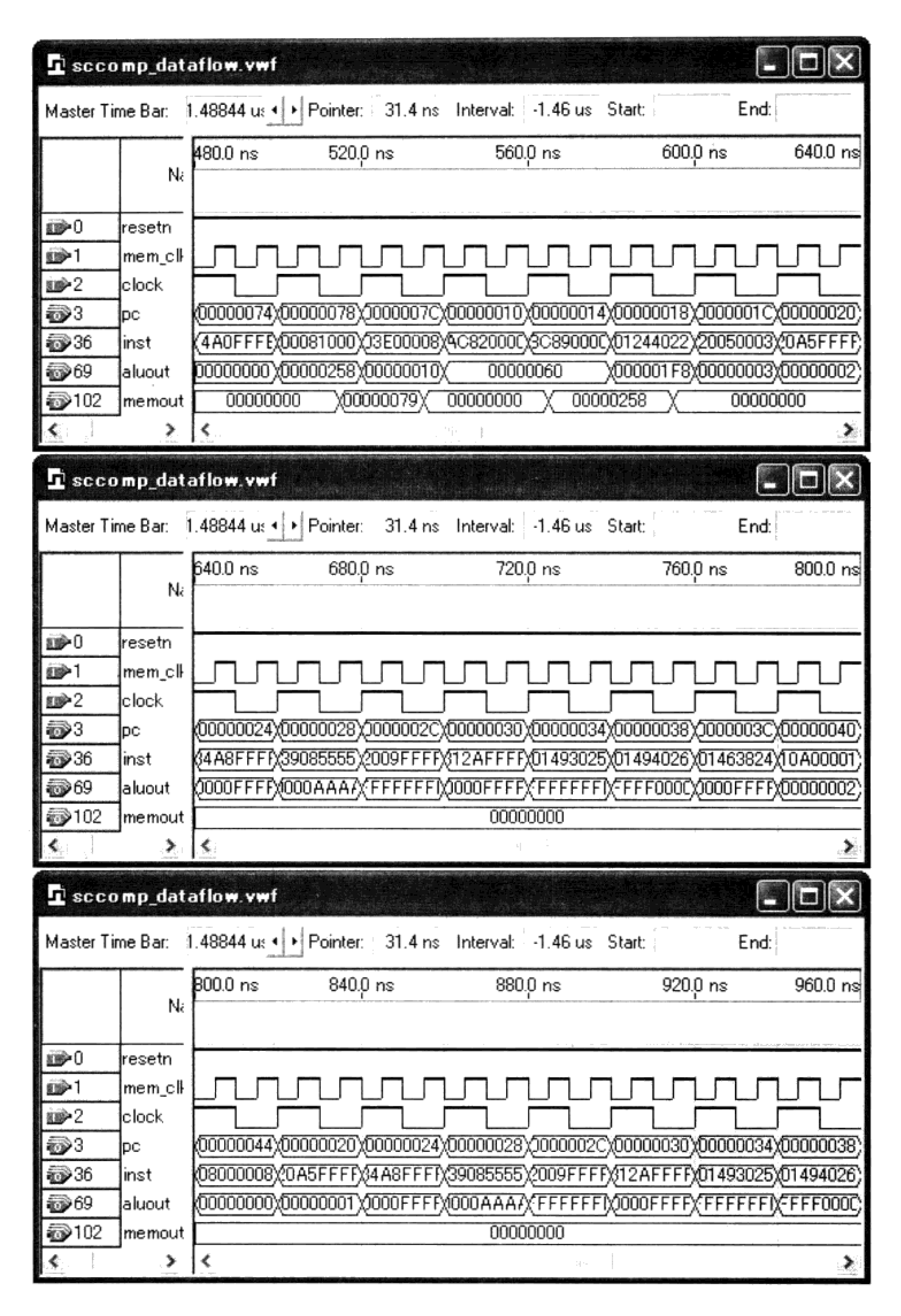


图 5.22 单周期 CPU 仿真波形图 (4 ~ 6)

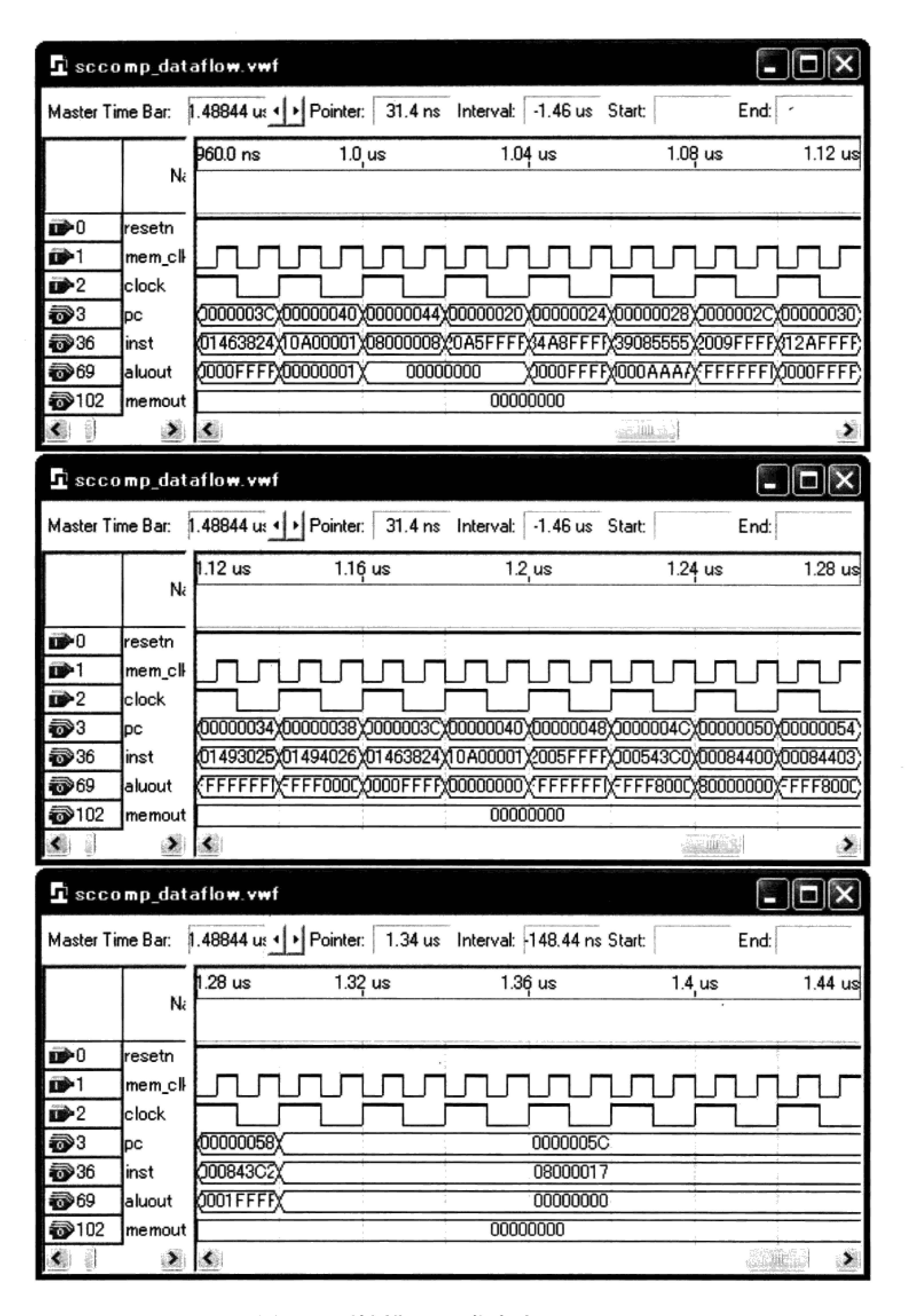


图 5.23　单周期 CPU 仿真波形图 (7 ~ 9)

```
    assign rom[5'h05] = 32'h8c890000; // (14)         lw   r9, 0(r4)
    assign rom[5'h06] = 32'h01244022; // (18)         sub  r8, r9, r4
    assign rom[5'h07] = 32'h20050003; // (1c)         addi r5, r0,  3
    assign rom[5'h08] = 32'h20a5ffff; // (20) loop2:  addi r5, r5, -1
    assign rom[5'h09] = 32'h34a8ffff; // (24)         ori  r8, r5, 0xffff
    assign rom[5'h0A] = 32'h39085555; // (28)         xori r8, r8, 0x5555
    assign rom[5'h0B] = 32'h2009ffff; // (2c)         addi r9, r0, -1
    assign rom[5'h0C] = 32'h312affff; // (30)         andi r10, r9, 0xffff
    assign rom[5'h0D] = 32'h01493025; // (34)         or   r6, r10, r9
    assign rom[5'h0E] = 32'h01494026; // (38)         xor  r8, r10, r9
    assign rom[5'h0F] = 32'h01463824; // (3c)         and  r7, r10, r6
    assign rom[5'h10] = 32'h10a00001; // (40)         beq  r5, r0, shift
    assign rom[5'h11] = 32'h08000008; // (44)         j    loop2
    assign rom[5'h12] = 32'h2005ffff; // (48) shift:  addi r5, r0, -1
    assign rom[5'h13] = 32'h000543c0; // (4c)         sll  r8, r5, 15
    assign rom[5'h14] = 32'h00084400; // (50)         sll  r8, r8, 16
    assign rom[5'h15] = 32'h00084403; // (54)         sra  r8, r8, 16
    assign rom[5'h16] = 32'h000843c2; // (58)         srl  r8, r8, 15
    assign rom[5'h17] = 32'h08000017; // (5c) finish: j    finish
    assign rom[5'h18] = 32'h00004020; // (60) sum:    add  r8, r0, r0
    assign rom[5'h19] = 32'h8c890000; // (64) loop:   lw   r9, 0(r4)
    assign rom[5'h1A] = 32'h20840004; // (68)         addi r4, r4,  4
    assign rom[5'h1B] = 32'h01094020; // (6c)         add  r8, r8, r9
    assign rom[5'h1C] = 32'h20a5ffff; // (70)         addi r5, r5, -1
    assign rom[5'h1D] = 32'h14a0fffb; // (74)         bne  r5, r0, loop
    assign rom[5'h1E] = 32'h00081000; // (78)         sll  r2, r8, 0
    assign rom[5'h1F] = 32'h03e00008; // (7c)         jr   r31
    assign inst = rom[a[6:2]];
endmodule
```

以下的数据存储器代码只用于仿真。

```
module scdatamem (clk, dataout, datain, addr, we, inclk, outclk);
    input  [31:0]  datain;
    input  [31:0]  addr;
    input          clk, we, inclk, outclk;
    output [31:0]  dataout;
    reg [31:0] ram [0:31];
    assign dataout = ram[addr[6:2]];
    always @ (posedge clk) begin
        if (we) ram[addr] = datain;
    end
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        ram[5'h14] = 32'h000000a3; // (50) data[0]   0 +  A3 =  A3
        ram[5'h15] = 32'h00000027; // (54) data[1]  A3 +  27 =  CA
```

```
        ram[5'h16] = 32'h00000079; // (58) data[2]  CA +  79 = 143
        ram[5'h17] = 32'h00000115; // (5c) data[3] 143 + 115 = 258
    end
endmodule
```

5.6 习题

1. 列出单周期 CPU 的优缺点并加以说明。
2. 用功能描述风格的 Verilog HDL 设计单周期 CPU。

   友情提示：不需要 ALU 控制码，使用 case 语句对指令一条条处理。以下是处理三条指令 add、lw 和 beq 的例子。注意 case 语句的用法。虽然控制信号声明为 reg 类型，但实际上将产生组合逻辑。

```
    wire [5:0]  opcode = inst[31:26];
    wire [4:0]  rs     = inst[25:21];
    wire [4:0]  rt     = inst[20:16];
    wire [4:0]  rd     = inst[15:11];
    wire [4:0]  sa     = inst[10:6];
    wire [5:0]  func   = inst[5:0];
    wire [15:0] imm    = inst[15:0];
    wire [25:0] addr   = inst[25:0];
    wire        sign   = inst[15];
    wire i_add = (opcode == 6'h00) & (func == 6'h20); // add
    wire i_lw  = (opcode == 6'h23);                   // lw
    wire i_beq = (opcode == 6'h04);                   // beq
    wire [31:0] pc_plus_4 = pc + 4;

    reg  [31:0] pc; // program counter
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) pc <= 0;
        else pc <= next_pc;
    end

    reg [31:0] regfile [1:31]; // registers
    always @ (posedge clock) begin
        if (wreg && (dest_rn != 0))
            if (i_lw)
                regfile[dest_rn] <= d_f_mem; // data from mem
            else
                regfile[dest_rn] <= ALU_out; // ALU output
    end

    wire [31:0] a = (rs == 0) ? 0 : regfile[rs];
    wire [31:0] b = (rt == 0) ? 0 : regfile[rt];
```

```
reg wreg,  wmem;     // write enables
reg [31:0] ALU_out; // ALU output
reg [4:0]  dest_rn; // destination register number
reg [31:0] next_pc; // next PC
always @(*) begin
    ALU_out = 0;
    dest_rn = rd;
    wreg    = 0;
    wmem    = 0;
    next_pc = pc_plus_4;
    case (1'b1)
        i_add: begin                                // add
            ALU_out = a + b;
            wreg    = 1;
        end
        i_lw: begin                                 // lw
            ALU_out = a + {{16{sign}},imm};
            dest_rn = rt;
            wreg    = 1;
        end
        i_beq: begin                                // beq
            if (a == b)
                next_pc = pc_plus_4 +
                          {{14{sign}},imm,2'b00};
        end
        default: ; // this is required
    endcase
end
```

3. 随便用什么语言写一个汇编器，把用本章实现的 20 条指令书写的汇编程序自动转换成 Altera 的 MIF 格式。提示：汇编器需要对汇编程序两遍扫描：第一遍建立地址标号的符号表。符号表中包括标号的名称和它们所对应的存储器地址。第二遍根据符号表中的地址生成条件转移或无条件跳转指令中的偏移量或地址。编译器也至少需要两遍扫描。
4. 上一习题再往前走一步，用高级语言程序仿真汇编程序的执行。允许输出是简单的文本方式。
5. 上一习题再往前走一步，使用 X (图形)。要有汇编器和反汇编器。
6. 上一习题再往前走一步，把 CPU 内部的硬件电路也以图形的方式显示出来。当然，各部分电路的仿真结果也要显示。
7. 还能再往前走吗？能，边仿真边做性能评价。不妨一试。
8. 实现所有 MIPS32 的整数指令，新加的指令要出现在测试程序中。MIPS32 的指令描述请参阅文献[22, 23, 24]。

第 6 章　异常和中断处理及其电路实现

在程序的执行过程当中，有时会出现预想不到的情况，比如计算结果的溢出、用于地址转换的 TLB 不命中等。另外，外部设备也可能向 CPU 发出信号，要求 CPU 为它提供服务。前者称为异常，后者称为中断。本章介绍 CPU 如何处理异常和中断，并给出相应的硬件设计及 Verilog HDL 代码。

6.1　异常和中断

本节讨论异常和中断的基本概念以及处理异常和中断的基本方法。

6.1.1　异常和中断的定义与类型

异常 (Exception) 和中断 (Interrupt) 是两种不可预知的事件，它们干扰程序的正常执行流程。异常来自于 CPU 的内部，比如做除法时除数为 0；而中断来自于 CPU 的外部，比如键盘中断。因此，如果我们分别称异常和中断为“内部同步异常”和“外部异步中断”，则更能反映出它们的准确含义。

当一个异常或一个中断出现时，CPU 应该停止当前程序的执行、转去执行预先准备好的程序去处理这个异常或中断。这个过程称为异常或中断响应。被执行的程序称为异常或中断处理程序，通常驻留在操作系统的内核。

依异常或中断的种类不同，异常或中断处理程序执行的最坏的结果可能是输出一些信息然后停机，最好的结果可能是做一些适当的处理然后恢复执行被停止的程序。图 6.1 示出了后一种情况的流程。

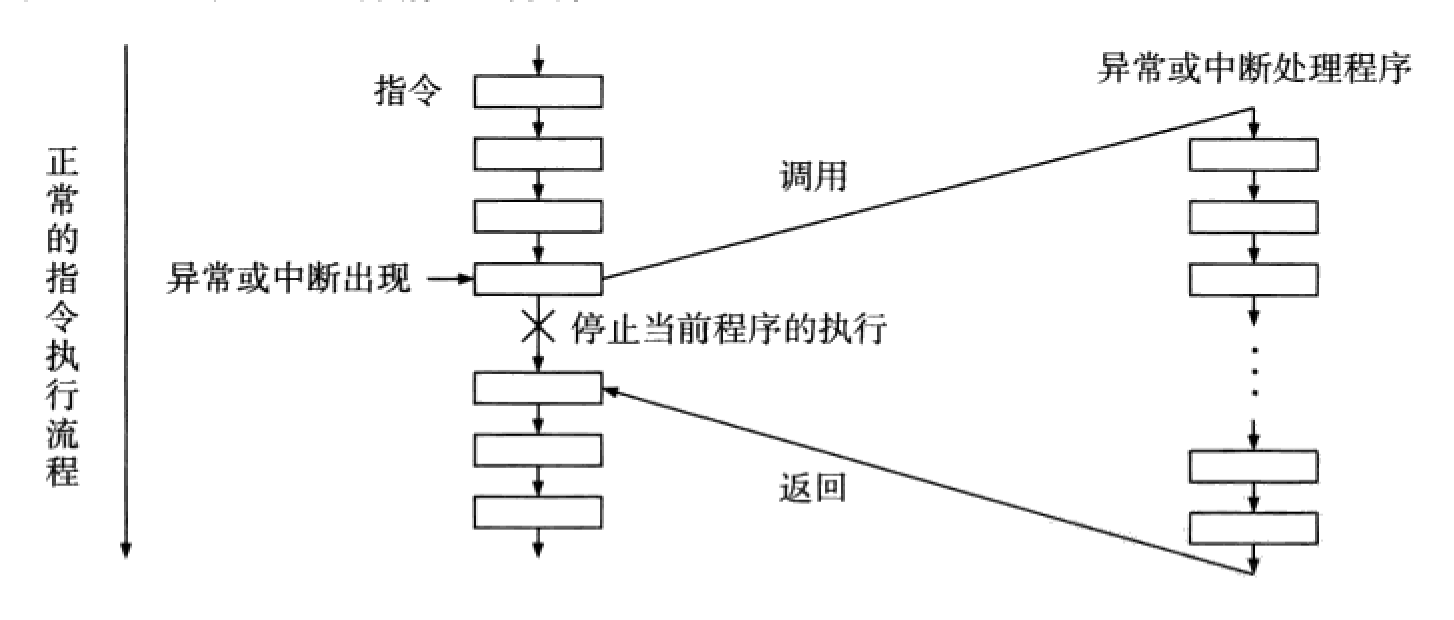


图 6.1　异常或中断的响应过程

溢出是一种常见的异常。溢出是指计算结果太大或太小而无法正确表示。例如 32 位补码表示的数据可以表示的数的范围是 $-2^{31} \sim +2^{31}-1$。如果计算结果大于 $+2^{31}-1$ 或小于 -2^{31}，则出现溢出。再比如浮点运算的结果太大或太小、除数为 0 或者对负数开方等，也会引起异常的出现。还有一种可能的情况就是取来的指令不

知道是什么意思，即指令操作码出错；或者取来的指令是一条意义明确的指令，但硬件没有实现它，而是通过调用子程序来实现它的功能；或者用户程序试图执行一条只有操作系统才有权利执行的特权指令。

其他的异常包括存储器地址没对准、存储器页失效 (CPU 要访问的内容不在存储器中，这是虚拟存储器管理中经常出现的正常情况)、存储器违法访问 (用户程序往操作系统区域写数据)、断点和单步执行 (为方便程序调试而设置) 等。

正常的系统调用有时也被归于异常。系统调用是一条指令，用户程序通过它来享受操作系统提供的服务。由于是一条指令，它便不是不可预知的事件，而是可以预知的。但对系统调用指令的处理与对通常的异常和中断的处理非常类似，因此我们也把它算作异常。

外部中断也有很多种情况，比如键盘中断、鼠标中断、打印机中断等。这些中断的共同特点是 I/O 设备要求 CPU 关照它们一下。另外还有计时器 (Timer) 的中断和硬件故障中断等。

6.1.2 查询中断和向量中断

图 6.1 给出了异常或中断的响应过程。那么一个问题出现了：如何从当前执行的程序跳转到异常或中断处理程序？有两种方法：查询中断 (Polled Interrupt) 和向量中断 (Vectored Interrupt)。

1. 查询中断

当异常或中断发生时，CPU 跳转到一个固定的地址，从那里开始查询到底出了什么状况，再转去执行相应的异常或中断处理程序。这个固定的地址可以用硬布线实现，这样就是真正意义上的固定了：CPU 芯片做好了以后，想修改这个地址也没有办法了。也可以稍微灵活一些，设置一个入口地址寄存器：当异常或中断发生时，把这个寄存器的内容打入程序计数器 PC。CPU 可以使用指令往这个寄存器写入不同的内容来修改这个入口地址。

但是，无论是硬布线还是使用入口寄存器，入口只有一个：不管现在发生的异常或中断是何种类型，CPU 都跳转到同一个地方开始执行程序。那么 CPU 如何知道已经发生的是什么异常或中断并如何做出相应的处理呢？这就需要有额外的信息了。我们可以设置一个专门的寄存器：当有异常或中断发生时，硬件能自动把发生源的信息记录到这个寄存器。CPU 可以在入口处读取这个寄存器，立刻就知道了是谁要引起“暴乱”，然后转移到专门的程序去对付它。MIPS CPU 采用的基本上就是这种查询中断方式。MIPS 称这个寄存器为 Cause 寄存器，见图 6.2。

31	30	29	28	27 ... 24	23	22	21 ... 16	15 ... 8	7	6 ... 2	1 0
	0			0			0	IP[7:0]	0	ExcCode	0

图 6.2 MIPS Cause 寄存器 (CP0 寄存器 13，只列出了与异常或中断有关的位)

MIPS 定义了很多这种类型的寄存器，统称为 CP0 (Coprocessor 0) 寄存器。Cause 寄存器中第 6 到第 2 位的 ExcCode 是引起异常或中断事件的代码、IP[7:0] 指出有哪些中断在门口等着接见呢。我们顺便展示一下 ExcCode，看看 MIPS CPU 能处理哪些异常，见表 6.1。有些异常的意义目前看不懂也没关系。

表 6.1 MIPS Cause 寄存器中 ExcCode 的定义

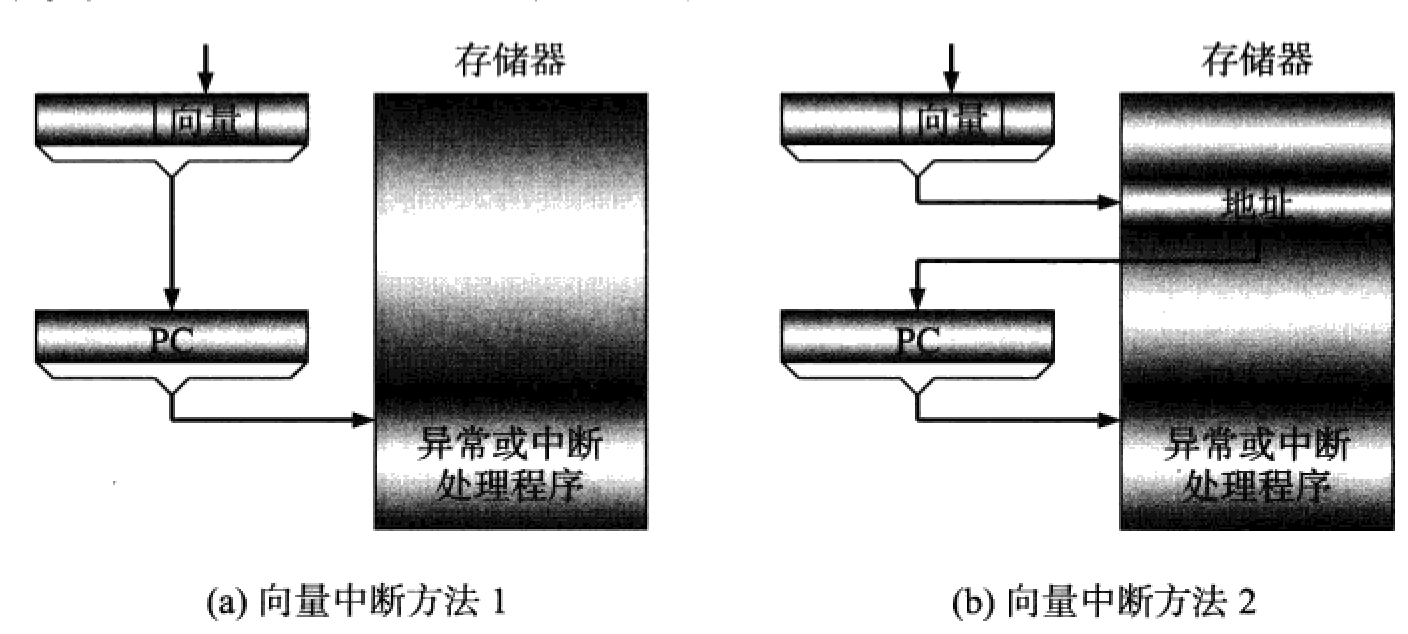
ExcCode 值	助记符	种类	描 述
0	Int	中断	中断 (IP[7:0] 指出中断源)
1	Mod	异常	TLB 项匹配但存储器页还没被写过 (存数据时)
2	TLBL	异常	TLB 项不匹配或无效 (取数据或取指令时)
3	TLBS	异常	TLB 项不匹配或无效 (存数据时)
4	AdEL	异常	存储器地址错 (取数据或取指令时)
5	AdES	异常	存储器地址错 (存数据时)
6	IBE	异常	总线错 (取指令时)
7	DBE	异常	总线错 (取数据或存数据时)
8	Sys	异常	执行系统调用指令
9	Bp	异常	执行断点指令
10	RI	异常	试图执行保留的指令
11	CpU	异常	协处理机不能用
12	Ov	异常	算术操作时结果溢出
13	Tr	异常	执行陷阱指令
14	–	–	保留
15	FPE	异常	浮点操作结果不正确
16 ～ 22	–	–	保留
23	WATCH	异常	虚拟地址与 Watch 寄存器的内容一致了
24	MCheck	异常	TLB 项匹配了不止一个，但不一致
25 ～ 29	–	–	保留
30	CacheErr	异常	Cache 出错
31	–	–	保留

以下的汇编程序演示 MIPS CPU 如何转到相应的程序去处理异常或中断。这里用到了一个事先准备好的散转表，其内容是处理各异常和中断程序的入口地址。散转表在存储器中的起始地址是 exc_tab_base，%hi(exc_tab_base) 和 %low(exc_tab_base) 分别是 exc_tab_base 的高 16 位和低 16 位。$k0 和 $k1 分别是寄存器 r26 和 r27。

```
lui  $k0, %hi(exc_tab_base)      # exc_table_base是散转表的起始地址
mfc0 $k1, C0_CAUSE               # 把Cause寄存器的内容送到寄存器r27
andi $k1, $k1, 0x7c              # 第6～2位是引起异常或中断的代码
add  $k0, $k0, $k1               # 找到散转表相应的地址
lw   $k0, %low(exc_tab_base)($k0) # 从散转表中取出入口地址
jr   $k0                         # 跳转到相应的程序去处理异常或中断
```

2. 向量中断

在向量中断方式中，引起异常或中断的“暴乱分子”自报家门，告诉警察“我是谁”。这个“我是谁”就是向量中断名称中的“向量”。然后有两种办法转移到异常或中断处理程序。先说第一种：见图 6.3(a)，CPU 硬件把这个向量放在中间，在它的左边和右边填上适当的数据，三部分合起来形成一个地址，把这个地址打入程序计数器 PC 中，这样就直接跳转到相应的入口去处理异常或中断。为什么还要在向量的两端填上数据再写入 PC？答案是“暴乱分子”没有那么多，只是“极少数”，否则就别干别的了。SUN SPARC CPU 采用的就是这种办法。



(a) 向量中断方法 1　　(b) 向量中断方法 2

图 6.3　向量中断

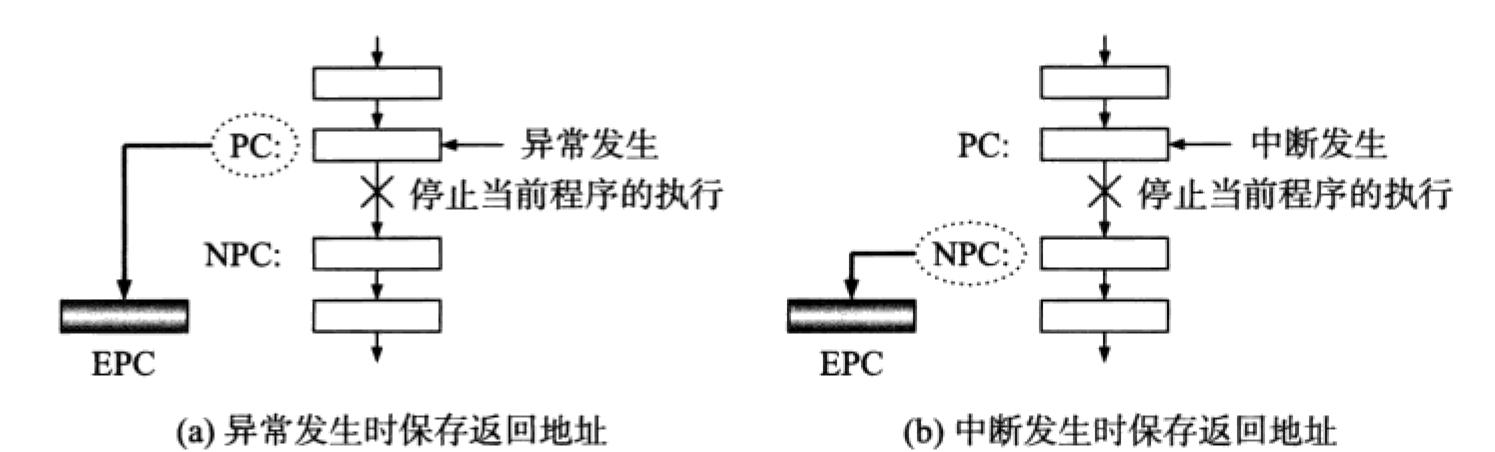
再说向量中断的第二种办法。见图 6.3(b)，CPU 硬件也是把这个向量放在中间，在它的左边和右边填上适当的数据，三部分合起来形成一个地址。下边请注意了：CPU 不是把这个地址打入程序计数器 PC 中，而是使用它访问存储器，把从存储器取来的数据打入 PC。Intel x86 CPU 大抵就是这么干的。

3. 中断返回

还有一件重要的事情没说呢。为使“受灾地区”恢复正常的生活秩序，我们必须要使 CPU 在处理完异常或中断后，返回到当初被打断的程序继续执行。怎么返回？要返回总要有地址啊。因此，在转向异常或中断处理程序时，不要忘记把返回地址保存到一个安全的地方。这个地方大有讲究。有些 CPU 的作法是把返回地址保存到一个通用寄存器中；有些 CPU 的作法是设置一个专门的寄存器，用于保存返回地址，比如 MIPS CPU 中的 EPC 寄存器；有些 CPU 则是把返回地址保存到存储器堆栈。既然是堆栈，那么栈顶指针应该能够自动调节。Intel x86 CPU 采用的就是这种办法。

MIPS CPU 所保存的“返回地址”有所不同：内部异常发生时，返回地址是引起异常的指令的地址 (PC)，这是因为引起异常的指令可能需要重新执行，见图 6.4(a)。

而外部中断发生时，返回地址是下一条指令的地址 (NPC)，因为外部中断发生时，当前 PC 所指的指令要执行完毕后，才转去执行异常或中断处理程序，见图 6.4(b)。



(a) 异常发生时保存返回地址　　(b) 中断发生时保存返回地址

图 6.4　MIPS CPU 保存返回地址

MIPS CPU 使用 eret 指令从异常或中断处理程序返回：把 EPC 的值写回 PC。因此，当某些引起异常的指令不需要重新执行时，返回之前要把 EPC 加 4，见下面的代码。当然，如果需要重新执行，就不用加 4 了。

```
mfc0  $k0, C0_EPC   # 把EPC寄存器的内容送到寄存器r26
addiu $k0, $k0, 4   # 加4
mtc0  $k0, C0_EPC   # 送回EPC寄存器
eret                # 从异常处理程序返回：PC <-- EPC
```

以上把异常或中断统统比喻为“暴乱分子”可能不太恰当，因为有些异常或中断是非常友善的。我们可以把有些异常或中断当成“上访”，有些当成是一般“市民”要求“政府”提供服务，而有些则是为“政府”提供服务。当然，硬件故障中断绝对属于暴乱，搞不好会使计算机系统瘫痪。

6.1.3　中断屏蔽和中断嵌套

执行中断处理程序时又出现中断请求怎么办？通常的做法是进入中断处理程序时就自动屏蔽中断，即“关中断”：来了中断 CPU 也不理睬。如果 CPU 想理睬呢？这就需要“开中断”。图 6.5 示出了 MIPS CPU Status 寄存器中有关中断屏蔽的控制位。IM[7:0] 中的每一位控制一个中断：为 0 时禁止中断，为 1 时允许中断 (不叫中断屏蔽，叫中断允许也许更贴切)；IE 是 IM[7:0] 的老板，老板说“关”，你们“开”也没用。注意，Status 寄存器名为“状态”，干的活儿却基本上是“控制”。总是感觉 MIPS 中有些名称和定义有点怪怪的，包括 TLB 的控制方式和有些指令的格式。

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–8	7	6	5	4	3	2	1	0
													IM[7:0]								IE

图 6.5　MIPS Status 寄存器 (CP0 寄存器 12，只列出了与中断屏蔽有关的位)

在处理当前的中断时又去响应新的中断，我们称其为中断嵌套 (Interrupt Nesting)，如图 6.6 所示。只有开了中断才会嵌套。开中断前要把“现场”保存好，尤其是专用寄存器或通用寄存器中的返回地址。如果返回地址在存储器堆栈中，则没有必要再保存它。为什么呢？

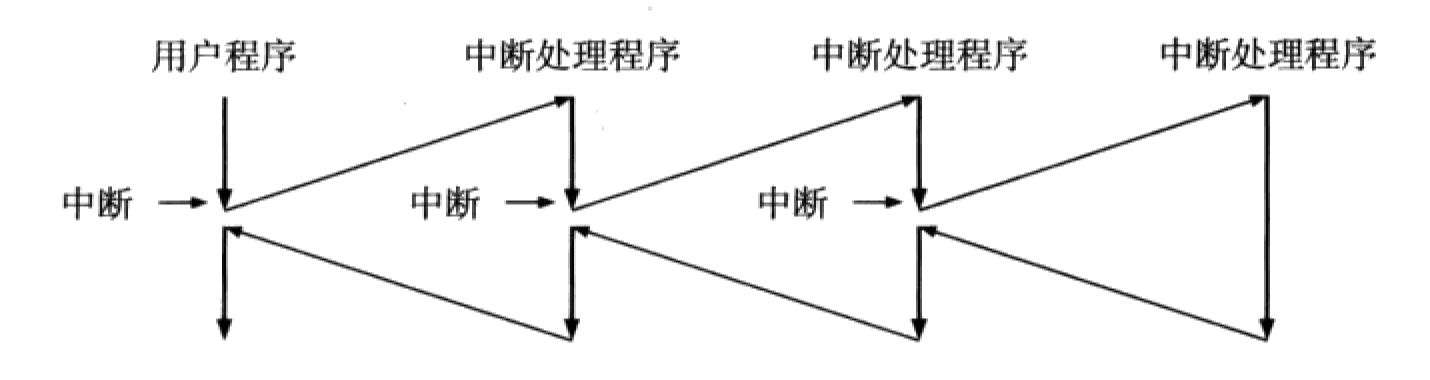


图 6.6 中断嵌套

6.1.4 中断优先级

如果有多个外部中断同时向 CPU 发出请求，CPU 响应哪一个呢？答案是响应优先级高的那一个。如何区分它们的优先级呢？方法是首先“排排队”，然后才“吃果果”。好像在第 2 章的习题中有一道题要你设计一个优先级编码器，不知道做了没有。把它用在中断控制器的设计中就能选择一个优先级较高的中断，见图 6.7 的中断控制器的示意图。图中 intr 是中断请求、inta 是中断确认、vector 是中断向量。

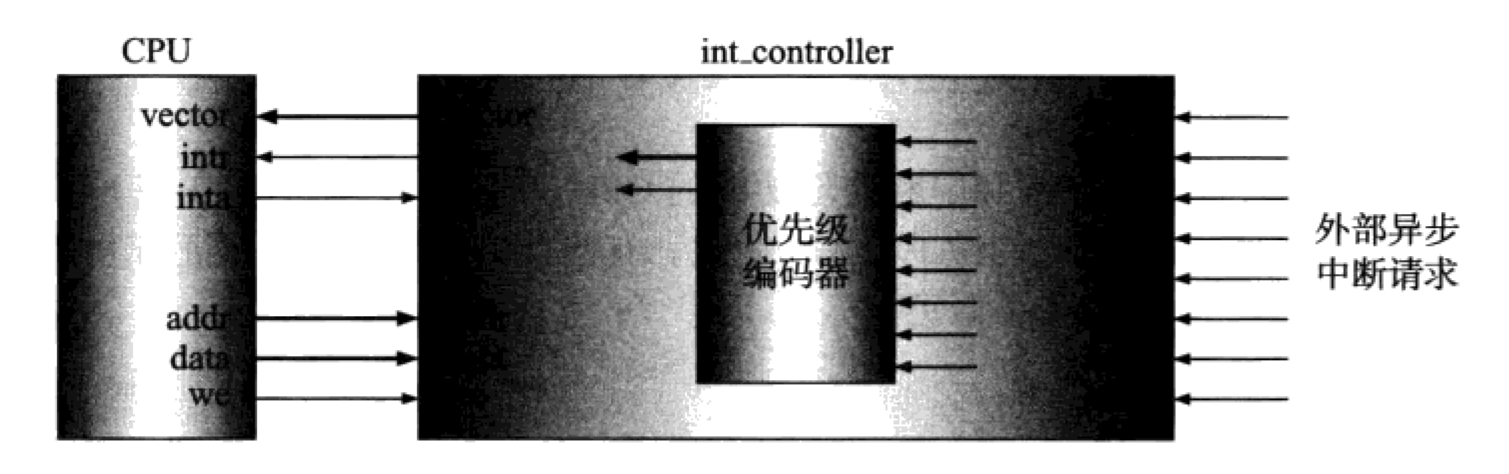


图 6.7 给 8 个中断源进行优先级编码从而选出优先级最高的中断请求

如果你真的没做那道题，没关系。图 6.8 有一道题正好等着你做呢：请设计图中的 DaisyChain 电路。图中 intr 是中断请求、inta 是中断确认。

千万要注意上边的非门一定要用漏极开路的 (TTL 中集电极开路的)，这样输出才能接在一起，否则电路就烧坏了。在 Quartus II 中使用漏极开路门的例子如下。关键器件是 opndrn (Open Drain)，它是一个漏极开路的缓冲器。这段 Verilog HDL 代码仅供参考，它只是演示 3 个输出如何接在一起。

```
module high_z_oc (in1,in2,in3,out1);
    input in1,in2,in3; // three input signals
```

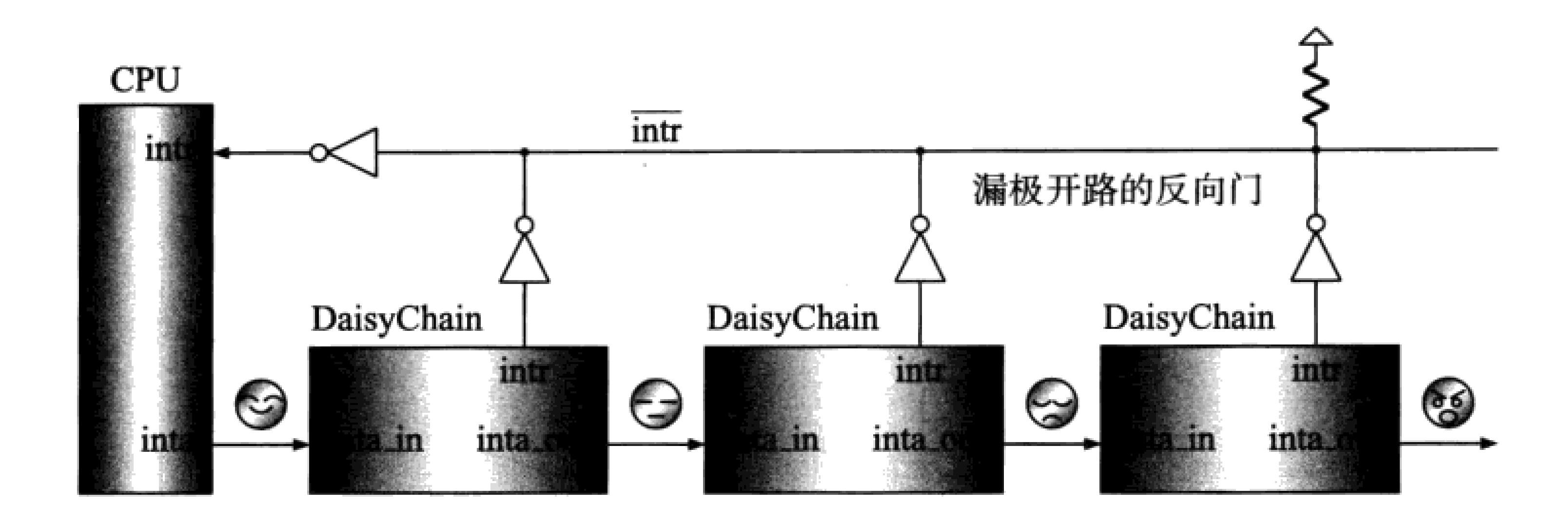


图 6.8 串行中断优先级查询 (Daisy-Chain)

```
    output out1;        // one output signal
    wire not_1_out,not_2_out,not_3_out;
    not not_1   (not_1_out,in1);
    not not_2   (not_2_out,in2);
    not not_3   (not_3_out,in3);
    // opndrn is an open-drain buffer
    opndrn oc1 (.in(not_1_out),.out(out1));
    opndrn oc2 (.in(not_2_out),.out(out1));
    opndrn oc3 (.in(not_3_out),.out(out1));
endmodule
```

6.2 带有异常和中断处理功能的 CPU 的设计

本节具体设计带有异常和中断处理功能的 CPU。本书的目的不是实现 MIPS 体系结构的所有功能，因此我们做了很大的简化。有些具体的实现也不保证与 MIPS 体系结构完全兼容。我们做如下假定。

1) 只有一个外部中断请求；
2) 只处理 3 个异常：结果溢出、出现未实现的指令以及系统调用；
3) 返回地址：异常时保存当前指令的地址，中断时保存下一条指令的地址；
4) 采用查询中断方式；
5) 响应异常或中断时把 Status 寄存器的内容左移 4 位自动关中断 (本书的做法)；
6) 中断返回时把 Status 寄存器的内容右移 4 位 (恢复原来的内容)。

6.2.1 异常和中断的处理过程以及相关的寄存器

CPU 响应一个内部同步异常或外部异步中断时，有以下 4 件事情要同时做。注意这些动作都是由硬件完成的，因此设计硬件时，我们要实现这些功能。

1) 把返回地址保存到 EPC 寄存器中，以便处理完异常或中断后返回；

2) 把引起异常或中断的原因自动记录到 Cause 寄存器 (ExcCode 域)；
3) 把 Status 寄存器中的中断屏蔽位左移 4 位 (自动关中断)；
4) 把一个固定的地址 (异常和中断处理程序的入口地址) 写入 PC。

如果是中断请求 intr，CPU 还要发出中断确认 inta。异常或中断处理程序可以读取 Cause 寄存器的内容来确认到底发生了什么事件，依此再跳转到相应的程序去对异常或中断进行处理。

处理完毕后，CPU 要返回到原来被打断的地方继续执行。返回靠执行 eret 指令实现。该指令要完成的动作有两个：把 EPC 的内容写回到 PC；与此同时，把 Status 寄存器中的内容右移 4 位 (恢复原来的中断屏蔽设置)。

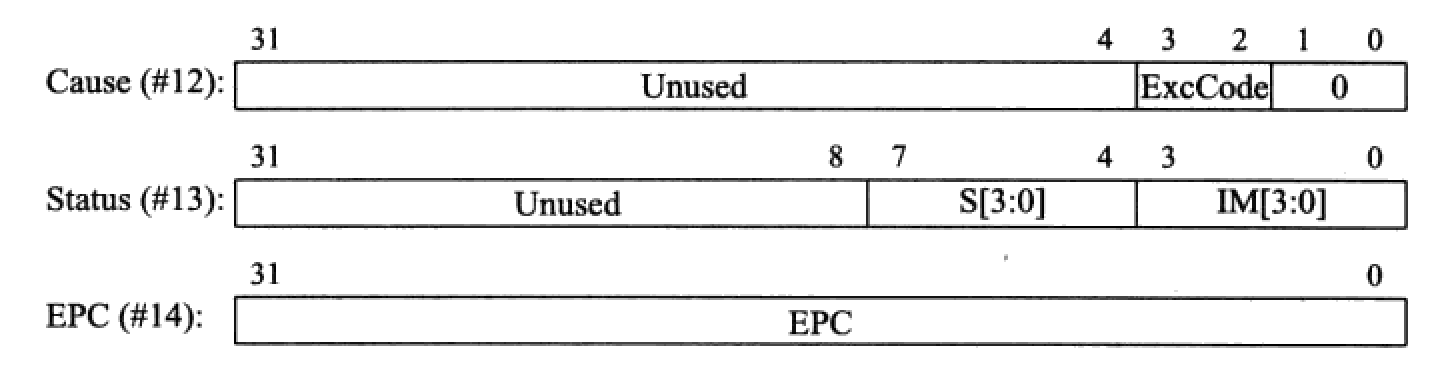


图 6.9 与异常和中断处理有关的 3 个寄存器

图 6.9 示出的是上面提到的 3 个寄存器。它们都属于 CP0 寄存器，每个寄存器都有一个号码，以便 CPU 使用 mfc0 或 mtc0 指令访问它。引起异常或中断的原因在 Cause 寄存器的 ExcCode 中，其编码见表 6.2。Status 寄存器中的 IM[3:0] 是 4 位屏蔽位，每位对应一个异常或中断。再说一遍，屏蔽位为 1 时允许异常或中断、为 0 时禁止。S[3:0] 是左移 4 位的 IM[3:0]。EPC 用于保存返回地址。

表 6.2 ExcCode 及 IM[3:0] 的定义

ExcCode 值	助记符	种类	屏蔽	描 述
0	Int	中断	IM[0]	外部中断
1	Sys	异常	IM[1]	执行系统调用指令 (多正常啊)
2	Unimpl	异常	IM[2]	试图执行没有实现的指令
3	Ov	异常	IM[3]	算术操作时结果溢出

6.2.2 与异常和中断有关的指令

引起算术计算结果溢出的指令有 3 条：add、sub 和 addi。这 3 条指令都是对带符号数进行计算。实际上对 ALU 来讲只有加减运算，因为 ALU 并不区分是 add 还是 addi。MIPS 还有对无符号数进行计算的指令，比如 addu。即使结果溢出，这类指令也不产生异常。表 6.3 列出了在什么情况下溢出 ($v = 1$)，其中 a[31] 和 b[31] 分别是补码表示的 32 位操作数 a 和 b 的最高位；r[31] 是计算结果的最高位。

表 6.3　加减操作的溢出

操作	aluc[3:0]	a[31]	b[31]	r[31]	v	解 释
ADD	x 0 0 0	0	0	1	1	正数加正数结果反而为负数
ADD	x 0 0 0	1	1	0	1	负数加负数结果反而为正数
SUB	x 1 0 0	0	1	1	1	正数减负数结果反而为负数
SUB	x 1 0 0	1	0	0	1	负数减正数结果反而为正数

根据表 6.3，我们直接写出溢出信号 v 的逻辑表达式 (Verilog HDL 格式)：

```
v = ~aluc[2] & ~a[31] & ~b[31] &  r[31] & ~aluc[1] & ~aluc[0] |
    ~aluc[2] &  a[31] &  b[31] & ~r[31] & ~aluc[1] & ~aluc[0] |
     aluc[2] & ~a[31] &  b[31] &  r[31] & ~aluc[1] & ~aluc[0] |
     aluc[2] &  a[31] & ~b[31] & ~r[31] & ~aluc[1] & ~aluc[0];
```

以上判断溢出的方法是最直接的，但也是最笨的方法。有一种非常简单的方法能得到非常简单的 v 的逻辑表达式，请读者试试看，能不能写出来。

CPU 可以使用 mfc0 rt, rd (Move from C0) 和 mtc0 rt, rd (Move to C0) 指令读写寄存器 Cause、Status 或 EPC。这两条指令中的 rt 是通用寄存器的号码，rd 是寄存器 Cause、Status 或 EPC 的号码。我们还需要有系统调用指令 syscall 和从异常或中断返回的指令 eret。以上 4 条指令的格式见图 6.10。

指令	31–26	25–21	20–16	15–11	10–0
mfc0 rt, rd	0 1 0 0 0 0	0 0 0 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0
mtc0 rt, rd	0 1 0 0 0 0	0 0 1 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0

指令	31–26	25–6	5–0
syscall	0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 1 1 0 0
eret	0 1 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 1 1 0 0 0

图 6.10　mfc0、mtc0、syscall 和 eret 指令的格式

6.2.3　带有异常和中断处理功能的 CPU 总体结构

经过以上的讨论，我们已经搞清楚了处理异常或中断的机制。在第 5 章，我们已经给出了单周期 CPU 的详细电路。在此基础上，我们加入异常和中断处理部分的电路。一种带有异常和中断处理功能的单周期 CPU 的总体电路如图 6.11 所示。

图 6.11 中新加了 3 个 CP0 寄存器 (Status、Cause 和 EPC) 以及对它们进行读写所需要的电路 (主要是多路选择器)。另外新加的电路是 PC 值的选择。除原来的之外，我们还应实现向异常或中断处理程序的跳转及返回。所有新加的电路所需的控制信号要由控制部件产生。

现把电路的工作过程说明如下。先说 mfc0 和 mtc0 两条指令的执行状况。mfc0 指令把 Status、Cause 或者 EPC 寄存器中的数据写入通用寄存器堆。寄存器数据的选

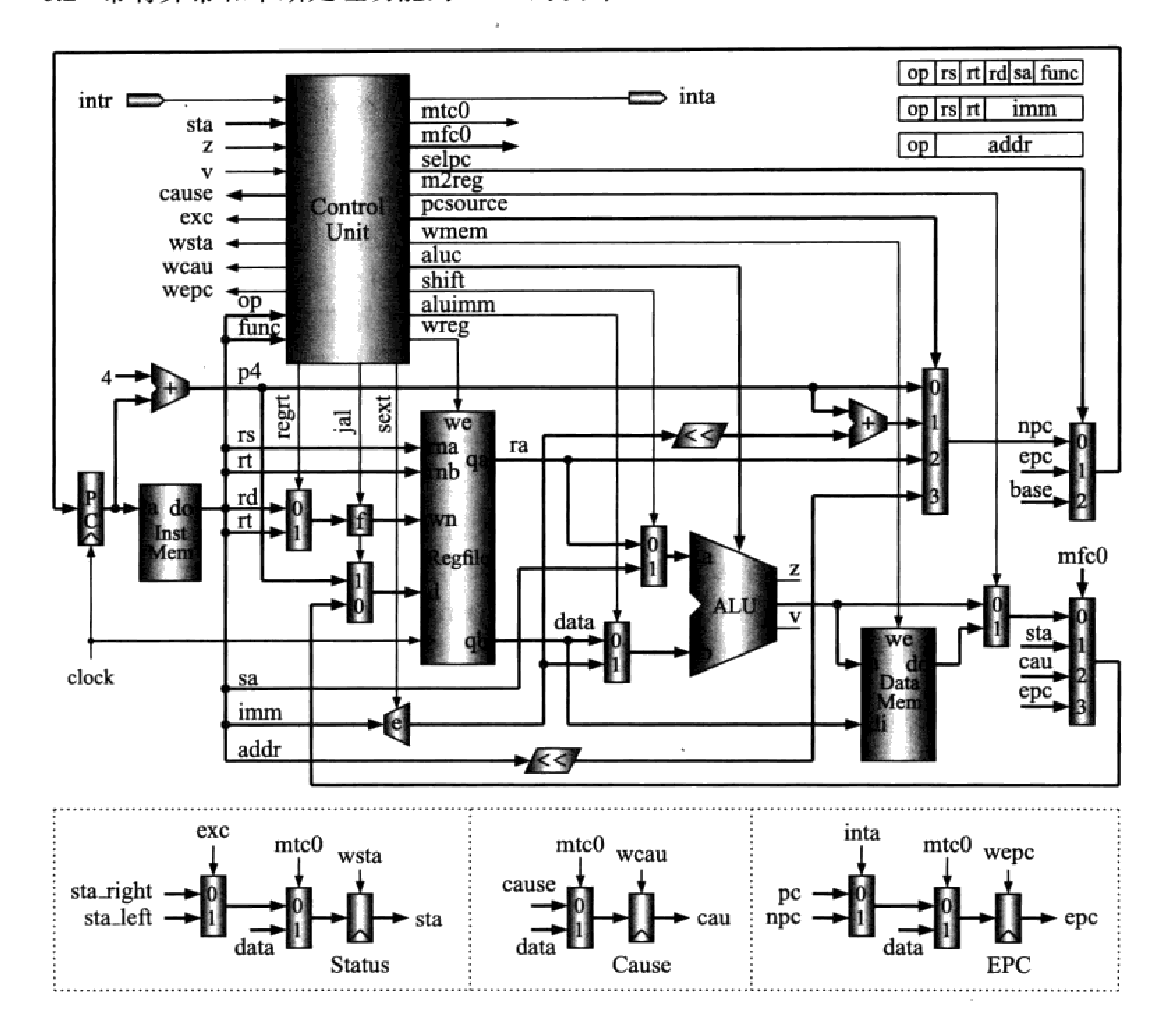


图 6.11 带有处理异常或中断功能的单周期 CPU

择由图中最右边的四选一多路器完成，选择信号为 mfc0。mtc0 指令把通用寄存器堆中的一个寄存器的内容写入Status、Cause 或者 EPC 寄存器。相应的写使能信号分别为 wsta、wcau 和 wepc。三个寄存器左边的二选一多路器的选择信号均为 wtc0。这时的 wtc0 应为 1，选择寄存器堆的 qb 的输出，即图中命名为 data 的数据。

再说异常或中断发生时的情况。如果是外部中断，输入管脚 intr 为 1；如果是溢出，ALU 的输出信号 v 为 1，送到控制部件；通过对当前指令译码，可知是否为未实现的指令或者 syscall 指令。总之控制部件知道是否出现了异常或中断。这时要使用 Status 寄存器的信息 sta 来查看相应的异常或中断是否被屏蔽。如果异常或中断出现了且没被禁止，控制部件生成 ExcCode，经由 cause 信号送给 Cause 寄存器左边的多路器。这时多路器的选择信号 mtc0 应为 0，以把 cause 数据写入 Cause 寄存器。如果读者读到这里问这时正在执行 wtc0 指令，而且也要往 Cause 寄存器中写数据怎么办，到底选哪一个？能提出这样的问题说明读者的段位已经相当高了。办法有一个：在执行 wtc0 指令前关掉所有的中断和异常。

不光是 Cause 寄存器，其他两个寄存器在异常或中断发生时也有数据要写入。往 EPC 中写入的是返回地址。中断时返回地址是下一条指令的地址，即图中的 npc；异常时返回地址是引起异常的指令的地址，即图中的 pc。选 pc 还是 npc 由 inta 决定。inta 是 CPU 的输出信号，意义为中断确认 (或中断回答或中断应答等，总之是英文 Interrupt Acknowledgement 的意思)。往 Status 寄存器写入的是 Status 寄存器左移 4 位后的内容，即图中的 sta_left。由于 Status 寄存器的最低 4 位是异常或中断屏蔽位，左移 4 位再写入相当于既保存了原来的屏蔽信息又禁止了异常或中断 (左移后最低 4 位填 0)。因此，左下角的多路器的选择信号 exc 应为 1，选择 sta_left。当然还要修改 PC，以实现向异常或中断处理程序的转移。这时， PC 数据输入端的多路器的选择信号 selpc 应为 2，选择 base 地址。这个 base 是“硬布线”的地址，也就是异常或中断处理程序的入口。

现在该讲如何从异常或中断返回了。CPU 执行 eret 指令实现返回。执行这条指令时，CPU 把 EPC 的内容写入 PC，因此 selpc 应为 1，以便选择 epc。与此同时，Status 寄存器的内容要右移 4 位，即图中的 sta_right，恢复它原来的真面目。怎么样？可以开始开发 CPU 的 Verilog HDL 代码了吧。

6.2.4 带有异常和中断处理功能的 CPU Verilog HDL 代码

CPU Verilog HDL 代码与第 5 章的代码有类似的架构，只是在最顶层加了两个信号：中断请求 intr (输入信号) 和中断确认 inta (输出信号)。最顶层代码调用 3 个模块：CPU 模块 (sccpu_intr)、指令存储器模块 (sci_intr) 和数据存储器模块 (scd_intr)。

```
module sc_interrupt (clock,resetn,inst,pc,aluout,memout,mem_clk,intr,inta);
    input clock,resetn,mem_clk,intr;
    output [31:0] inst,pc,aluout,memout;
    output inta;
    wire [31:0] data;
    wire wmem;
    sccpu_intr cpu (clock,resetn,inst,memout,pc,wmem,aluout,data,intr,inta);
    sci_intr imem (pc,inst);
    scd_intr dmem (clock,memout,data,aluout,wmem,mem_clk,mem_clk);
endmodule
```

以下是 CPU 模块的代码。控制部件和 ALU 重新设计过了：

```
module sccpu_intr (clock,resetn,inst,mem,pc,wmem,alu,data,intr,inta);
    input  [31:0] inst,mem;
    input  clock,resetn,intr;
    output [31:0] pc,alu,data;
    output wmem,inta;
    parameter   EXC_BASE = 32'h00000008; // = base = BASE
    wire [31:0] p4,bpc,npc,adr,ra,alua,alub,res,alu_mem;
    wire [3:0]  aluc;
    wire [4:0]  reg_dest,wn;
```

```
    wire [1:0]  pcsource;
    wire zero,wmem,wreg,regrt,m2reg,shift,aluimm,jal,sext,overflow;
    wire [31:0] sa = {27'b0,inst[10:6]};
    wire [31:0] offset = {imm[13:0],inst[15:0],1'b0,1'b0};
    sccu_intr cu (inst[31:26],inst[25:21],inst[15:11],inst[5:0],zero,wmem,
        wreg,regrt,m2reg,aluc,shift,aluimm,pcsource,jal,sext,
        intr,inta,overflow,sta,cause,exc,wsta,wcau,wepc,mtc0,mfc0,selpc);
    wire e = sext & inst[15];
    wire [15:0] imm = {16{e}};
    wire [31:0] immediate = {imm,inst[15:0]};
    dff32 ip (next_pc,clock,resetn,pc); // next_pc
    cla32 pcplus4 (pc,32'h4,1'b0,p4);
    cla32 br_adr  (p4,offset,1'b0,adr);
    wire [31:0] jpc = {p4[31:28],inst[25:0],1'b0,1'b0};
    mux2x32 alu_b  (data,immediate,aluimm,alub);
    mux2x32 alu_a  (ra,sa,shift,alua);
    mux2x32 result (alu,mem,m2reg,alu_mem);
    mux2x32 link   (alu_mem_c0,p4,jal,res); // alu_mem_c0
    mux2x5  reg_wn (inst[15:11],inst[20:16],regrt,reg_dest);
    assign  wn = reg_dest | {5{jal}}; // jal: r31 <-- p4;
    mux4x32 nextpc (p4,adr,ra,jpc,pcsource,npc);
    regfile rf (inst[25:21],inst[20:16],res,wn,wreg,clock,resetn,ra,data);
    alu_ov al_unit (alua,alub,aluc,alu,zero,overflow);
```

CPU 模块新加的与异常或中断有关的电路：3 个寄存器和 7 个多路器：

```
    wire exc,wsta,wcau,wepc,mtc0;
    wire [31:0] sta,cau,epc,sta_in,cau_in,epc_in,
                sta_l1_a0,epc_l1_a0,cause,alu_mem_c0,next_pc;
    wire [1:0]  mfc0,selpc;
    dffe32  c0_Status  (sta_in,clock,resetn,wsta,sta); // Status register
    dffe32  c0_Cause   (cau_in,clock,resetn,wcau,cau); // Cause register
    dffe32  c0_EPC     (epc_in,clock,resetn,wepc,epc); // EPC register
    mux2x32 sta_l1 (sta_l1_a0,data,mtc0,sta_in);       // for Status
    mux2x32 sta_l2 ({4'h0,sta[31:4]},{sta[27:0],4'h0},exc,sta_l1_a0);
    mux2x32 cau_l1 (cause,data,mtc0,cau_in);           // for Cause
    mux2x32 epc_l1 (epc_l1_a0,data,mtc0,epc_in);       // for EPC
    mux2x32 epc_l2 (pc,npc,inta,epc_l1_a0);
    mux4x32 irq_pc (npc,epc,EXC_BASE,32'h0,selpc,next_pc); // for PC
    mux4x32 fromc0 (alu_mem,sta,cau,epc,mfc0,alu_mem_c0);  // for mfc0
endmodule
```

控制部件，重点是与异常或中断有关的译码：

```
module sccu_intr (op,op1,rd,func,z,wmem,wreg,regrt,m2reg,aluc,
       shift,aluimm,pcsource,jal,sext,
       intr,inta,ov,sta,cause,exc,wsta,wcau,wepc,mtc0,mfc0,selpc);
input [5:0] op,func;
input [4:0] op1,rd;
```

```
input  z;
output wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem;
output [3:0] aluc;
output [1:0] pcsource;
```

与异常或中断有关的信号：

```
input intr,ov;
input [31:0] sta; // IM[3:0] : ov,unimpl,sys,int
output inta,exc,wsta,wcau,wepc,mtc0;
output [1:0] mfc0,selpc;
output [31:0] cause;
```

区别异常和中断的类型，4 种：外部中断、系统调用、未实现的指令以及溢出。

```
wire overflow = ov & (i_add | i_sub | i_addi);
assign inta   = int_int;
wire   int_int = sta[0] & intr;
wire   exc_sys = sta[1] & i_syscall;
wire   exc_uni = sta[2] & unimplemented_inst;
wire   exc_ovr = sta[3] & overflow;
assign exc     = int_int | exc_sys | exc_uni | exc_ovr;
```

产生 ExcCode：00：外部中断、01：系统调用、10：未实现的指令、11：溢出。

```
wire   ExcCode0 = i_syscall | overflow;
wire   ExcCode1 = unimplemented_inst | overflow;
assign cause = {28'h0,ExcCode1,ExcCode0,2'b00};
```

产生 3 个寄存器的写使能信号：

```
assign mtc0  = i_mtc0;
assign wsta  = exc | mtc0 & rd_is_status | i_eret;
assign wcau  = exc | mtc0 & rd_is_cause;
assign wepc  = exc | mtc0 & rd_is_epc;
```

执行 mfc0 指令时选择寄存器：00：原来的、01：Status、10：Cause、11：EPC。

```
wire   rd_is_status = (rd == 5'd12); // cp0 Status register
wire   rd_is_cause  = (rd == 5'd13); // cp0 Cause register
wire   rd_is_epc    = (rd == 5'd14); // cp0 EPC register
assign mfc0[0] = i_mfc0 & rd_is_status | i_mfc0 & rd_is_epc;
assign mfc0[1] = i_mfc0 & rd_is_cause  | i_mfc0 & rd_is_epc;
```

PC 的选择：00：原来的、01：EPC、10：异常或中断处理程序的入口。

```
assign selpc[0] = i_eret;
assign selpc[1] = exc;
```

对新加的 4 条指令译码以及确定是否为未实现的指令：

```
wire c0_type= ~op[5] & op[4] &~op[3] &~op[2] &~op[1] &~op[0];
wire i_mfc0 = c0_type &~op1[4] &~op1[3] &~op1[2] &~op1[1] &~op1[0];
wire i_mtc0 = c0_type &~op1[4] &~op1[3] & op1[2] &~op1[1] &~op1[0];
wire i_eret = c0_type & op1[4] &~op1[3] &~op1[2] &~op1[1] &~op1[0]&
        ~func[5] & func[4] & func[3] &~func[2] &~func[1] &~func[0];
wire i_syscall = r_type &~func[5] &~func[4] & func[3] & func[2] &
                ~func[1] &~func[0];
wire unimplemented_inst = ~(i_mfc0 | i_mtc0 | i_eret | i_syscall |
     i_add | i_sub  | i_and  | i_or   | i_xor  | i_sll  | i_srl  |
     i_sra | i_jr   | i_addi | i_andi | i_ori  | i_xori | i_lw   |
     i_sw  | i_beq  | i_bne  | i_lui  | i_j    | i_jal);
```

以下与第 5 章的内容基本相同，只是顾及了 mfc0 指令：

```
wire r_type = ~|op;
wire i_add = r_type& func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
wire i_sub = r_type& func[5]&~func[4]&~func[3]&~func[2]& func[1]&~func[0];
wire i_and = r_type& func[5]&~func[4]&~func[3]& func[2]&~func[1]&~func[0];
wire i_or  = r_type& func[5]&~func[4]&~func[3]& func[2]&~func[1]& func[0];
wire i_xor = r_type& func[5]&~func[4]&~func[3]& func[2]& func[1]&~func[0];
wire i_sll = r_type&~func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
wire i_srl = r_type&~func[5]&~func[4]&~func[3]&~func[2]& func[1]&~func[0];
wire i_sra = r_type&~func[5]&~func[4]&~func[3]&~func[2]& func[1]& func[0];
wire i_jr  = r_type&~func[5]&~func[4]& func[3]&~func[2]&~func[1]&~func[0];
wire i_addi = ~op[5] &~op[4] & op[3] &~op[2] &~op[1] &~op[0];
wire i_andi = ~op[5] &~op[4] & op[3] & op[2] &~op[1] &~op[0];
wire i_ori  = ~op[5] &~op[4] & op[3] & op[2] &~op[1] & op[0];
wire i_xori = ~op[5] &~op[4] & op[3] & op[2] & op[1] &~op[0];
wire i_lw   =  op[5] &~op[4] &~op[3] &~op[2] & op[1] & op[0];
wire i_sw   =  op[5] &~op[4] & op[3] &~op[2] & op[1] & op[0];
wire i_beq  = ~op[5] &~op[4] &~op[3] & op[2] &~op[1] &~op[0];
wire i_bne  = ~op[5] &~op[4] &~op[3] & op[2] &~op[1] & op[0];
wire i_lui  = ~op[5] &~op[4] & op[3] & op[2] & op[1] & op[0];
wire i_j    = ~op[5] &~op[4] &~op[3] &~op[2] & op[1] &~op[0];
wire i_jal  = ~op[5] &~op[4] &~op[3] &~op[2] & op[1] & op[0];
assign wreg = i_add | i_sub | i_and| i_or  | i_xor| i_sll| i_srl| i_sra|
              i_addi| i_andi| i_ori| i_xori| i_lw| i_lui| i_jal| i_mfc0;
assign regrt   = i_addi| i_andi| i_ori| i_xori| i_lw | i_lui| i_mfc0;
assign jal     = i_jal;
assign m2reg   = i_lw;
assign shift   = i_sll | i_srl | i_sra;
assign aluimm  = i_addi| i_andi| i_ori| i_xori| i_lw | i_lui| i_sw;
assign sext    = i_addi| i_lw  | i_sw | i_beq | i_bne;
assign aluc[3] = i_sra;
assign aluc[2] = i_sub| i_or| i_srl| i_sra| i_ori| i_lui;
assign aluc[1] = i_xor| i_sll| i_srl| i_sra| i_xori| i_beq| i_bne| i_lui;
assign aluc[0] = i_and| i_or| i_sll| i_srl| i_sra| i_andi| i_ori;
assign wmem    = i_sw;
```

```
assign pcsource[1] = i_jr| i_j| i_jal;
assign pcsource[0] = i_beq & z| i_bne &~z| i_j| i_jal;
endmodule
```

带有溢出标志 v 的 ALU。v 的产生方法很笨：

```
module alu_ov (a,b,aluc,r,z,v);
    input [31:0] a,b;                                // aluc[3:0]
    input [3:0]  aluc;                               //
    output [31:0] r;                                 // x 0 0 0  ADD
    output z,v;                                      // x 1 0 0  SUB
    wire [31:0] d_and = a & b;                       // x 0 0 1  AND
    wire [31:0] d_or  = a | b;                       // x 1 0 1  OR
    wire [31:0] d_xor = a ^ b;                       // x 0 1 0  XOR
    wire [31:0] d_lui = {b[15:0],16'h0};             // x 1 1 0  LUI
    wire [31:0] d_and_or  = aluc[2]? d_or  : d_and;  // 0 0 1 1  SLL
    wire [31:0] d_xor_lui = aluc[2]? d_lui : d_xor;  // 0 1 1 1  SRL
    wire [31:0] d_as,d_sh;                           // 1 1 1 1  SRA
    addsub32 as32 (a,b,aluc[2],d_as);
    shift shifter (b,a[4:0],aluc[2],aluc[3],d_sh);
    mux4x32 select (d_as,d_and_or,d_xor_lui,d_sh,aluc[1:0],r);
    assign z = ~|r;
    assign v = ~aluc[2] & ~a[31] & ~b[31] &  r[31] & ~aluc[1] & ~aluc[0] |
               ~aluc[2] &  a[31] &  b[31] & ~r[31] & ~aluc[1] & ~aluc[0] |
                aluc[2] & ~a[31] &  b[31] &  r[31] & ~aluc[1] & ~aluc[0] |
                aluc[2] &  a[31] & ~b[31] & ~r[31] & ~aluc[1] & ~aluc[0];
endmodule
```

指令存储器，测试程序在 sci_intr.mif 中：

```
module sci_intr (a,inst);
    input [31:0] a;
    output [31:0] inst;
    lpm_rom  lpm_rom_component (.address(a[7:2]),.q(inst));
    defparam lpm_rom_component.lpm_width    = 32,
             lpm_rom_component.lpm_widthad  = 6,
             lpm_rom_component.lpm_numwords = "unused",
             lpm_rom_component.lpm_file     = "sci_intr.mif",
             lpm_rom_component.lpm_indata   = "unused",
             lpm_rom_component.lpm_outdata  = "unregistered",
             lpm_rom_component.lpm_address_control = "unregistered";
endmodule
```

数据存储器，测试数据在 scd_intr.mif 中：

```
module scd_intr (clk, dataout, datain, addr, we, inclk, outclk);
    input  [31:0]  datain;
    input  [31:0]  addr;
    input          clk, we, inclk, outclk;
```

```
    output [31:0]  dataout;

    wire           write_enable = we & ~clk;
    lpm_ram_dq ram (.data(datain),.address(addr[6:2]),
                    .we(write_enable),.inclock(inclk),
                    .outclock(outclk),.q(dataout));
    defparam   ram.lpm_width    = 32;
    defparam   ram.lpm_widthad = 5;
    defparam   ram.lpm_indata   = "registered";
    defparam   ram.lpm_outdata = "registered";
    defparam   ram.lpm_file     = "scd_intr.mif";
    defparam   ram.lpm_address_control = "registered";
endmodule
```

6.3 CPU 的异常与中断测试

6.3.1 测试程序和测试数据

以下给出测试程序和测试数据。注意在测试程序中，我们主要是演示异常或中断处理程序的进入和返回，而没有对异常或中断本身进行任何处理。

1. 指令存储器的内容 sci_intr.mif

```
DEPTH = 64;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
[0..3F] : 00000000; % Range--Every address from 0 to 3F = 00000000     %
```

系统复位 (Reset) 时的入口：0x00000000。

```
              %     reset:                                             %
 0: 0800001d; % (00)  j    start             # entry on reset          %
 1: 00000000; % (04)  nop                    #                         %
```

异常或中断处理程序入口：0x00000008。查表并跳转。跳转表在数据存储器中。

```
              %     EXC_BASE:                # exception handler       %
 2: 401a6800; % (08)  mfc0 r26, C0_CAUSE     # read cp0 Cause reg      %
 3: 335b000c; % (0c)  andi r27, r26, 0xc     # get ExcCode, 2 bits here %
 4: 8f7b0020; % (10)  lw   r27, j_table(r27)# get address from table  %
 5: 00000000; % (14)  nop                    #                         %
 6: 03600008; % (18)  jr   r27               # jump to that address    %
 7: 00000000; % (1c)  nop                    #                         %
```

外部中断处理程序入口：0x00000030。什么也没干就返回了。

```
            %     int_entry:               # 0. interrupt handler      %
 c: 00000000; % (30)  nop                  # deal with interrupt here %
 d: 42000018; % (34)  eret                 # return from interrupt    %
 e: 00000000; % (38)  nop                  #                          %
```

系统调用指令的入口：0x0000003c。EPC 加 4 后就返回了。

```
            %     sys_entry:               # 1. SysCall handler       %
 f: 00000000; % (3c)  nop                  # do something here        %
            %     epc_plus4:               #                          %
10: 401a7000; % (40)  mfc0  r26, C0_EPC    # get EPC                  %
11: 235a0004; % (44)  addi  r26, r26, 4    # EPC + 4                  %
12: 409a7000; % (48)  mtc0  r26, C0_EPC    # EPC <-- EPC + 4          %
13: 42000018; % (4c)  eret                 # return from exception    %
14: 00000000; % (50)  nop                  #                          %
```

未实现的指令的入口：0x00000054。EPC 加 4 后就返回了。

```
            %     uni_entry:               # 2. Unimpl. inst. handler %
15: 00000000; % (54)  nop                  # do something here        %
16: 08000010; % (58)  j     epc_plus4      # return                   %
17: 00000000; % (5c)  nop                  #                          %
```

溢出处理程序的入口：0x00000068。EPC 加 4 后就返回了。

```
            %     ovf_entry:               # 3. Overflow handler      %
1a: 00000000; % (68)  nop                  # do something here        %
1b: 08000010; % (6c)  j     epc_plus4      # return                   %
1c: 00000000; % (70)  nop                  #                          %
```

开中断：

```
            %     start:                   #                          %
1d: 2008000f; % (74)  addi r8, r0, 0xf     # IM[3:0] <-- 1111         %
1e: 40886000; % (78)  mtc0 r8, C0_STATUS   # exc/intr enable          %
```

测试溢出：

```
1f: 8c080048; % (7c)  lw    r8, 0x48(r0)   # try overflow exception   %
20: 8c09004c; % (80)  lw    r9, 0x4c(r0)   # caused by add            %
            %     Ov:                      #                          %
21: 01094020; % (84)  add   r9, r9, r8     # overflow                 %
22: 00000000; % (88)  nop                  #                          %
```

测试系统调用：

```
            %     Sys:                     #                          %
23: 0000000c; % (8c)  syscall              #                          %
24: 00000000; % (90)  nop                  #                          %
```

测试未实现的指令：

```
              %     Unimpl:                 #                           %
25: 0128001a; % (94)  div  r9, r8           # div, but not implemented %
26: 00000000; % (98)  nop                   #                           %
```

测试外部中断：在此期间由外部送来中断信号 intr。

```
              %     Int:                    #                           %
27: 34040050; % (9c)  ori  r4, r1, 0x50     # address of data[0]        %
28: 20050004; % (a0)  addi r5, r0,  4       # counter                   %
29: 00004020; % (a4)  add  r8, r0, r0       # sum <-- 0                 %
              %     loop:                   #                           %
2a: 8c890000; % (a8)  lw   r9, 0(r4)        # load data                 %
2b: 20840004; % (ac)  addi r4, r4,  4       # address + 4               %
2c: 01094020; % (b0)  add  r8, r8, r9       # sum                       %
2d: 20a5ffff; % (b4)  addi r5, r5, -1       # counter - 1               %
2e: 14a0fffb; % (b8)  bne  r5, r0, loop     # finish?                   %
2f: 00000000; % (bc)  nop                   #                           %
              %     finish:                 #                           %
30: 08000030; % (c0)  j    finish           # dead loop                 %
END ;
```

2. 数据存储器的内容 scd_intr.mif

```
DEPTH = 32;            % Memory depth and width are required    %
WIDTH = 32;            % Enter a decimal number                 %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless    %
                       % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
[0..1F] : 00000000; % Range--Every address from 0 to 1F = 00000000 %
```

跳转表：

```
   % address table for internal exception and external interrupt %
   8 : 00000030; % (20) int_entry # 0. address for interrupt     %
   9 : 0000003c; % (24) sys_entry # 1. address for Syscall       %
   a : 00000054; % (28) uni_entry # 2. address for Unimpl. inst. %
   b : 00000068; % (2c) ovf_entry # 3. address for Overflow      %
```

为测试溢出而准备的数据：

```
  12 : 00000002; % (48) for testing overflow                     %
  13 : 7fffffff; % (4c) 2 + max_int --> overflow                 %
  14 : 000000A3; % (50) data[0]   0 +  A3 =  A3                  %
  15 : 00000027; % (54) data[1]  A3 +  27 =  CA                  %
  16 : 00000079; % (58) data[2]  CA +  79 = 143                  %
  17 : 00000115; % (5C) data[3] 143 + 115 = 258                  %
END ;
```

6.3.2 CPU 异常及中断处理测试结果及说明

图 6.12 ~ 图 6.19 给出了带有异常和中断处理功能的 CPU 在执行上述测试程序时产生的波形。从图 6.12 看出，复位时 CPU 从 0x00000000 地址开始执行。因为 0x00000000 地址的指令是 j 指令，跳转到 0x00000074 去执行。开中断后，在 PC = 0x00000084 处的 add 指令产生溢出，从而进入异常或中断处理程序的总入口 0x00000008。即，图 6.12 是执行以下 7 条指令的波形。

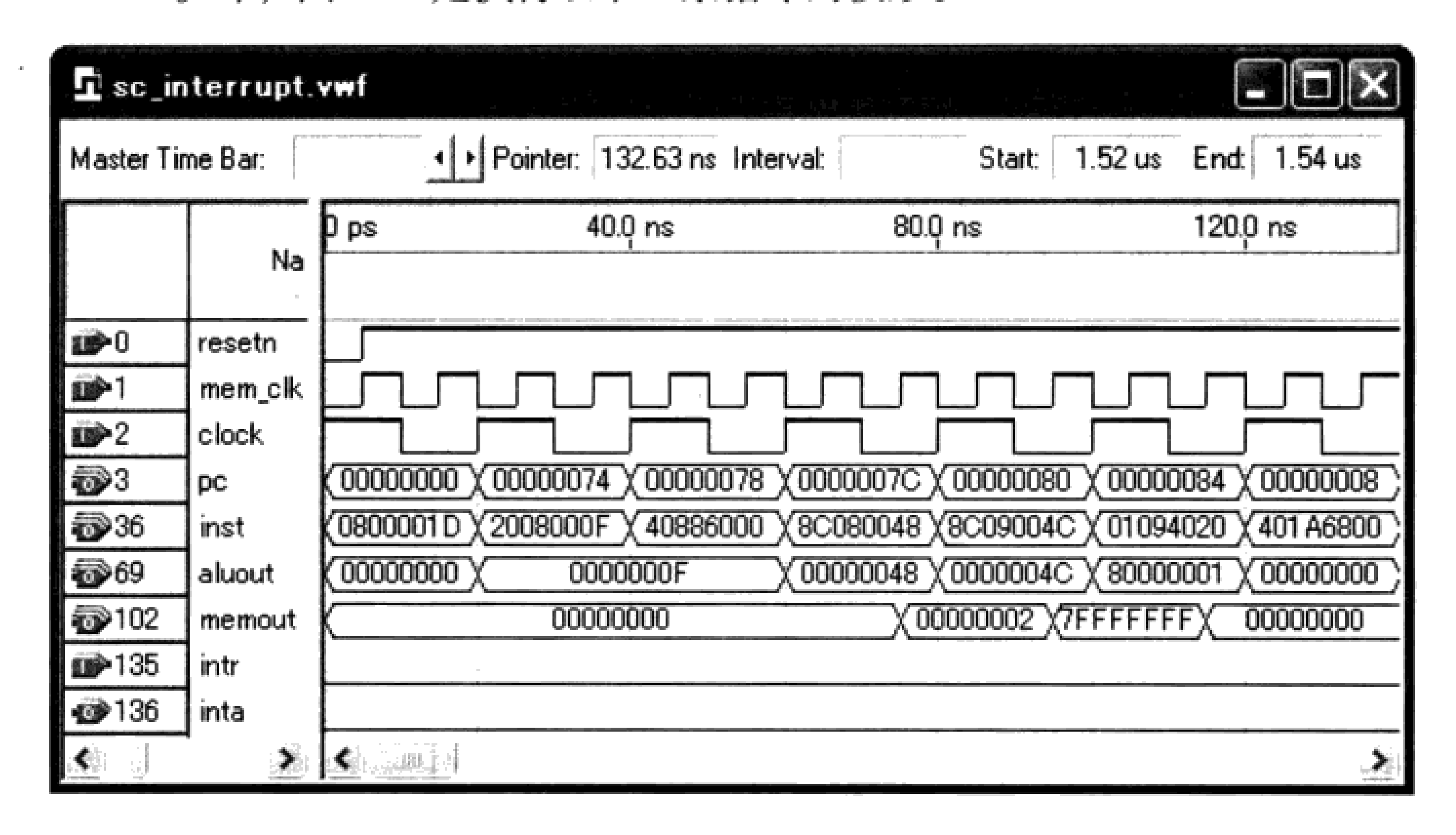


图 6.12 单周期 CPU 异常或中断仿真波形图 (1)

```
 0: 0800001d; % (00) reset:      j    start           %
1d: 2008000f; % (74) start:      addi r8, r0, 0xf     %
1e: 40886000; % (78)             mtc0 r8, C0_STATUS   %
1f: 8c080048; % (7c)             lw   r8, 0x48(r0)    %
20: 8c09004c; % (80)             lw   r9, 0x4c(r0)    %
21: 01094020; % (84) Ov:         add  r9, r9, r8      %
 2: 401a6800; % (08) EXC_BASE:   mfc0 r26, C0_CAUSE   %
```

图 6.13 是图 6.12 的继续。主要工作是检查异常类型并转入相应的处理程序的入口处。由于本次的异常是溢出，因此转入 0c00000068。然后从 0x0000006c 转去做 EPC 加 4。图 6.13 是执行以下 7 条指令的波形。

```
 3: 335b000c; % (0c)             andi r27, r26, 0xc   %
 4: 8f7b0020; % (10)             lw   r27, j_table(r27) %
 5: 00000000; % (14)             nop                  %
 6: 03600008; % (18)             jr   r27             %
1a: 00000000; % (68) ovf_entry:  nop                  %
1b: 08000010; % (6c)             j     epc_plus4      %
10: 401a7000; % (40) epc_plus4:  mfc0  r26, C0_EPC    %
```

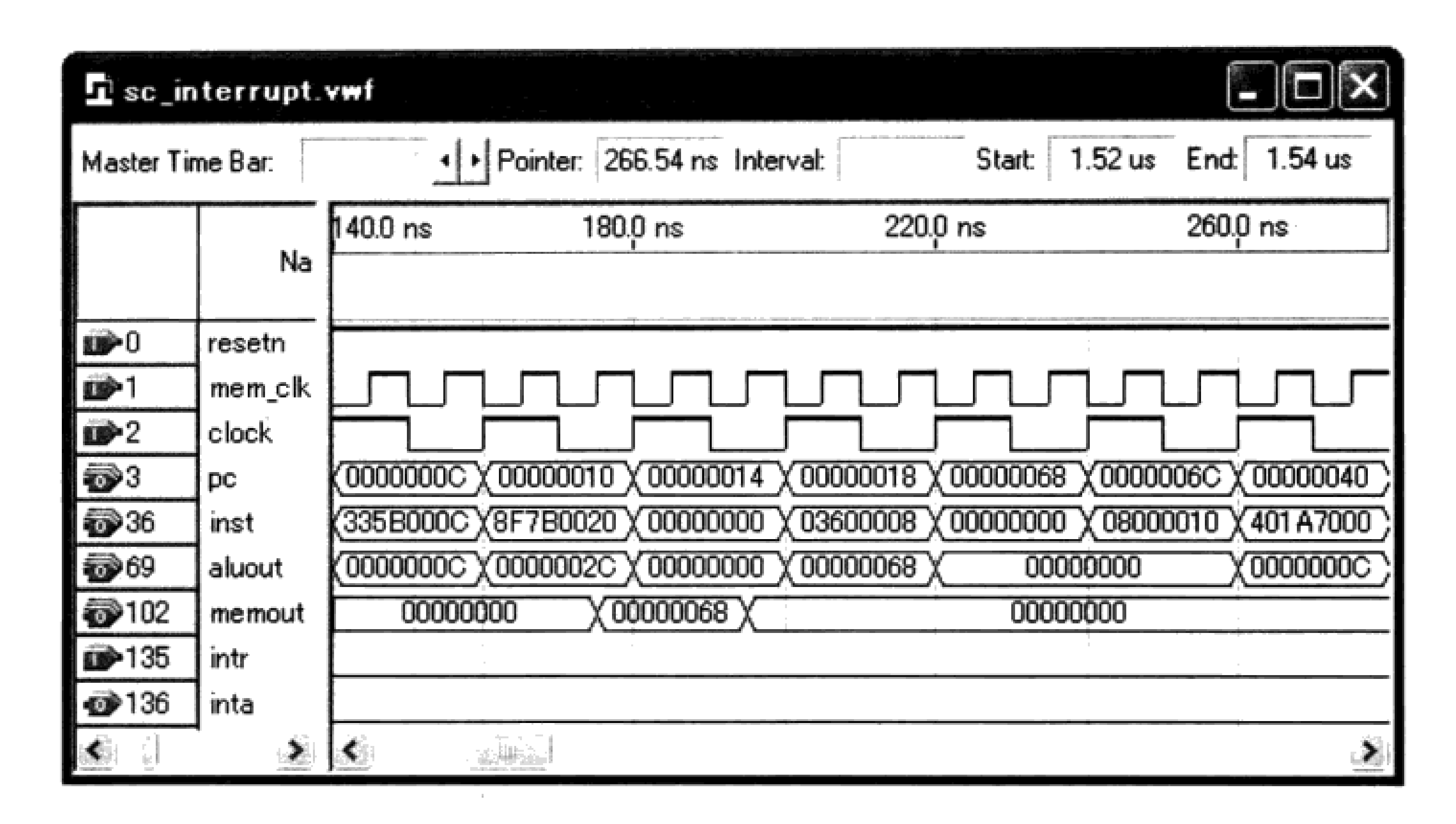


图 6.13 单周期 CPU 异常或中断仿真波形图 (2)

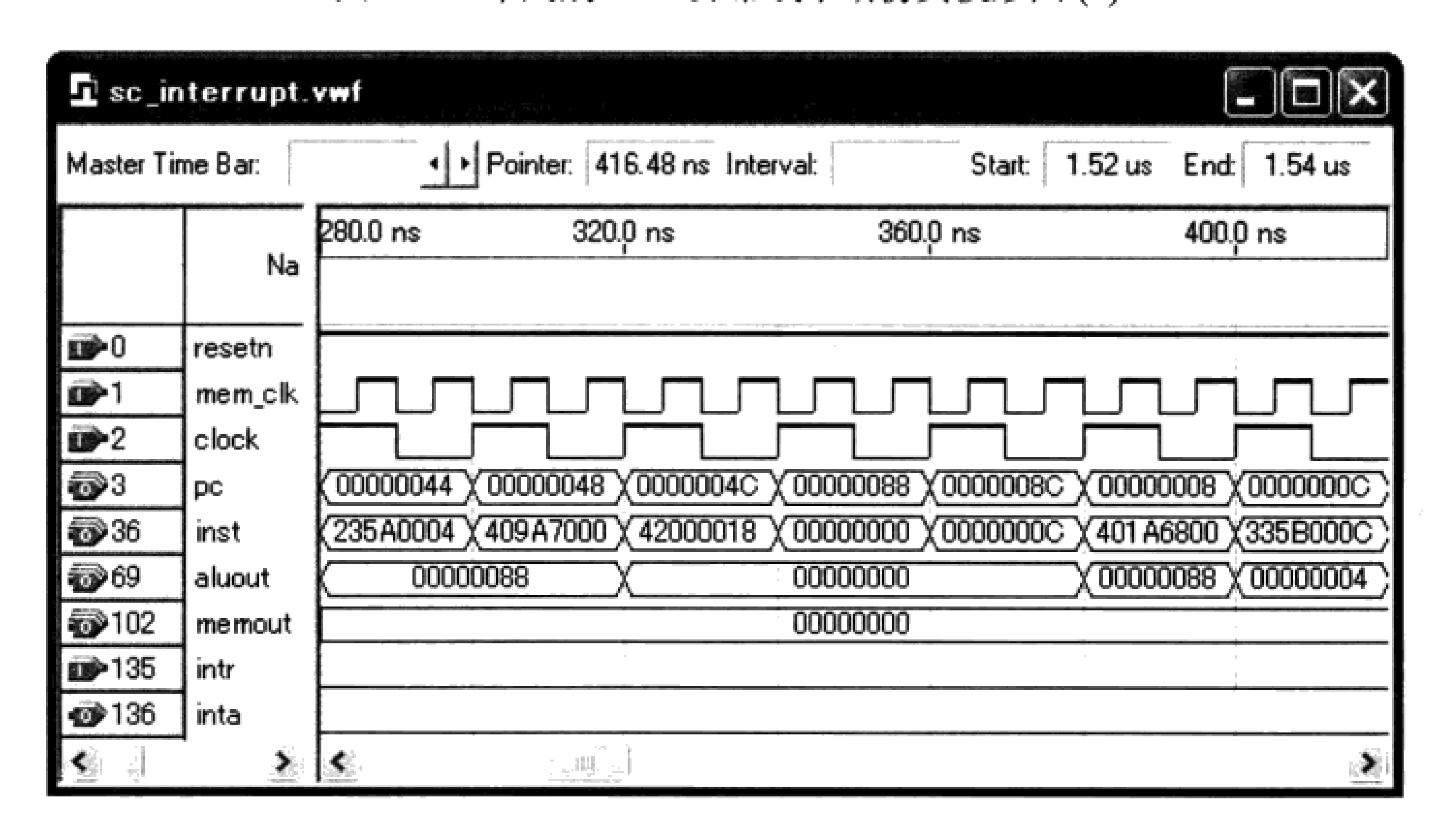


图 6.14 单周期 CPU 异常或中断仿真波形图 (3)

图 6.14 的指令完成 EPC 加 4，然后返回。返回后执行系统调用 syscall 指令，又进入异常或中断处理程序的总入口处。图 6.14 是执行以下 7 条指令的波形。

```
11: 235a0004; % (44)              addi   r26, r26, 4      %
12: 409a7000; % (48)              mtc0   r26, C0_EPC      %
13: 42000018; % (4c)              eret                    %
22: 00000000; % (88)              nop                     %
23: 0000000c; % (8c) Sys:         syscall                 %
 2: 401a6800; % (08) EXC_BASE:    mfc0 r26, C0_CAUSE      %
```

```
 3: 335b000c; % (0c)              andi r27, r26, 0xc     %
```

图 6.15 演示进入系统调用异常的处理程序。它是执行以下 7 条指令的波形。

```
 4: 8f7b0020; % (10)              lw   r27, j_table(r27) %
 5: 00000000; % (14)              nop                    %
 6: 03600008; % (18)              jr   r27               %
 f: 00000000; % (3c) sys_entry:   nop                    %
10: 401a7000; % (40) epc_plus4:   mfc0  r26, C0_EPC      %
11: 235a0004; % (44)              addi  r26, r26, 4      %
12: 409a7000; % (48)              mtc0  r26, C0_EPC      %
```

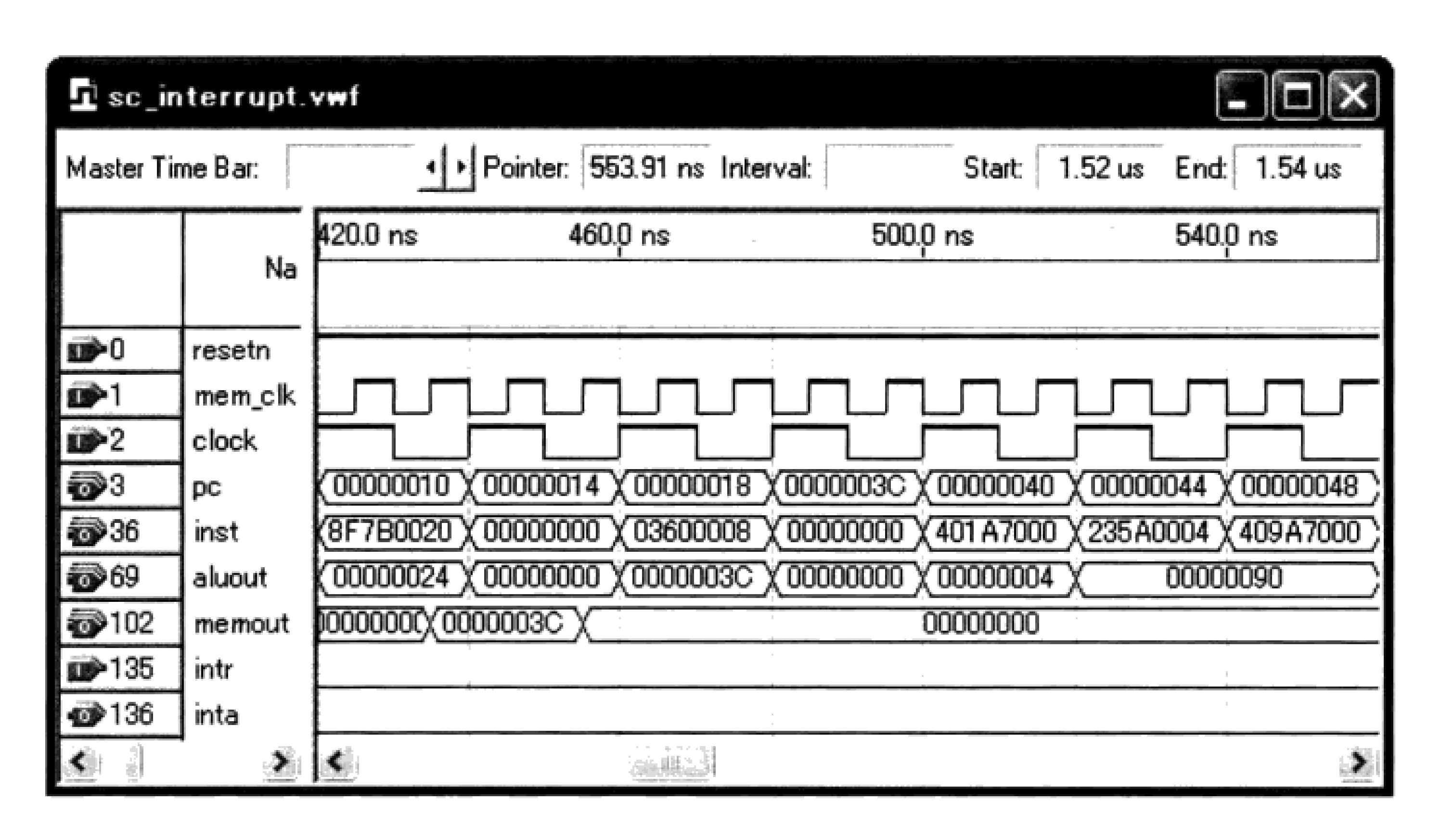


图 6.15 单周期 CPU 异常或中断仿真波形图 (4)

图 6.16 演示的是执行 div r9, r8。这是一条合法的 MIPS 的整数除法指令，但我们的 CPU 硬件没有实现它，因此产生未实现的指令异常。也是进入异常或中断处理程序的总入口。图 6.16 示出执行以下 7 条指令的波形。

```
13: 42000018; % (4c)              eret                   %
24: 00000000; % (90)              nop                    %
25: 0128001a; % (94) Unimpl:      div  r9, r8            %
 2: 401a6800; % (08) EXC_BASE:    mfc0 r26, C0_CAUSE     %
 3: 335b000c; % (0c)              andi r27, r26, 0xc     %
 4: 8f7b0020; % (10)              lw   r27, j_table(r27) %
 5: 00000000; % (14)              nop                    %
```

图 6.17 演示进入未实现的指令异常的处理程序，执行以下 7 条指令。

```
 6: 03600008; % (18)              jr   r27               %
15: 00000000; % (54) uni_entry:   nop                    %
16: 08000010; % (58)              j     epc_plus4        %
```

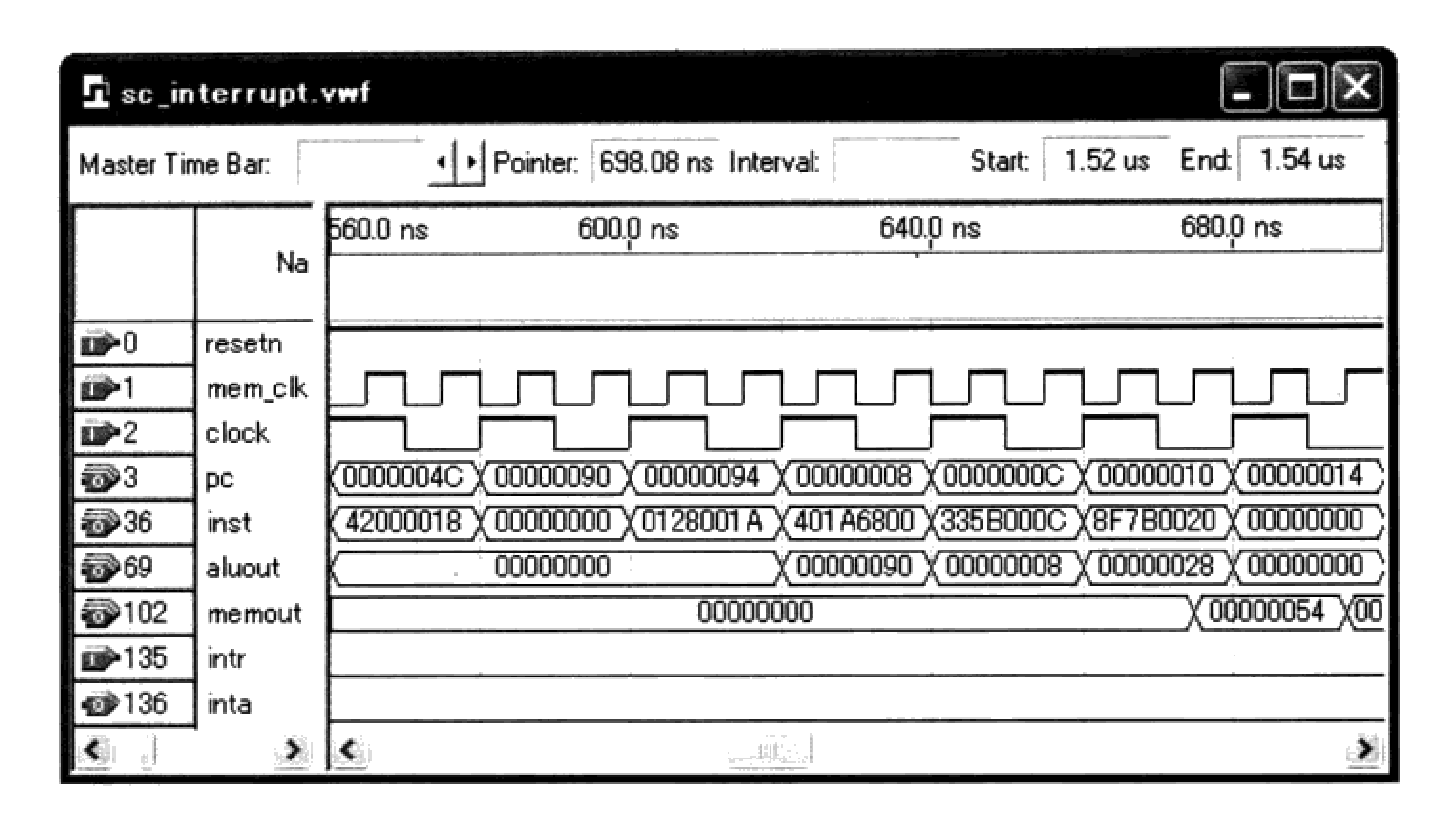


图 6.16 单周期 CPU 异常或中断仿真波形图 (5)

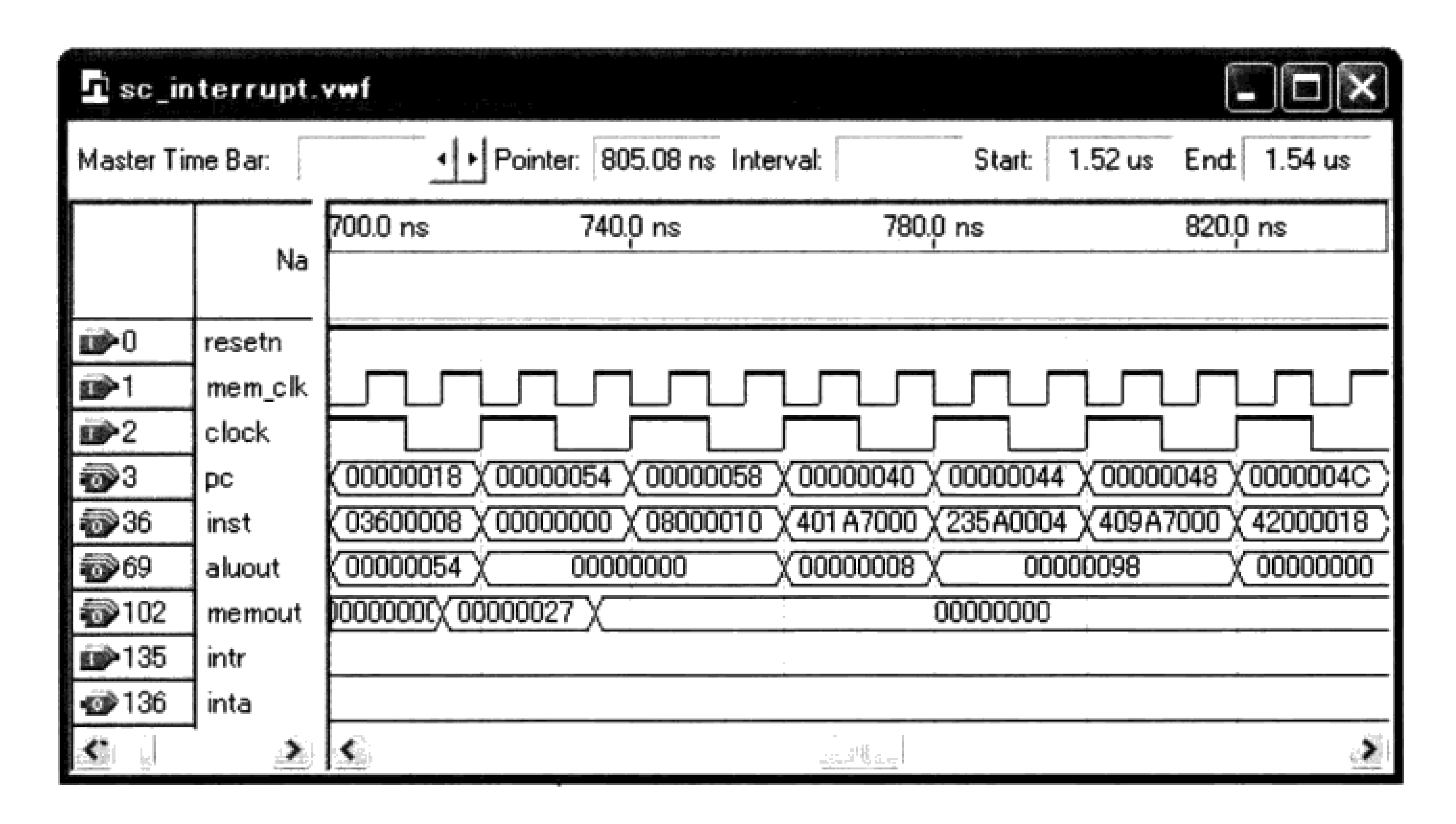


图 6.17 单周期 CPU 异常或中断仿真波形图 (6)

```
10: 401a7000; % (40) epc_plus4:  mfc0  r26, C0_EPC      %
11: 235a0004; % (44)             addi  r26, r26, 4      %
12: 409a7000; % (48)             mtc0  r26, C0_EPC      %
13: 42000018; % (4c)             eret                   %
```

三种异常都演示过了。图 6.18 演示的是最后一种：外部中断。外部中断是异步的，你不知道它什么时候来。在我们的仿真中，通过硬性置 intr 信号为 1 来产生外部中断。从图中可以看出，中断出现在 CPU 正在执行 PC = 0x000000AC 处的指令的

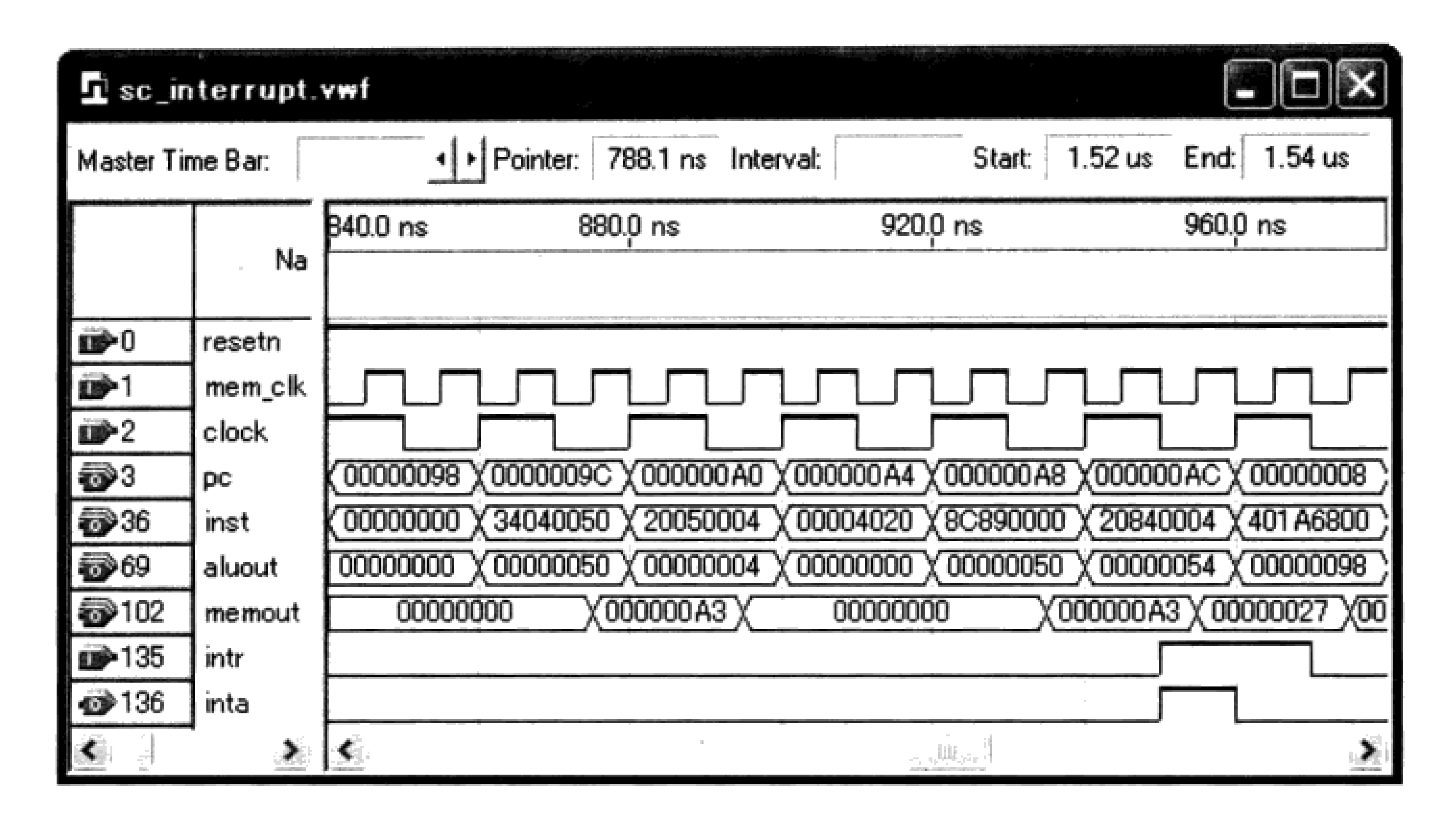


图 6.18　单周期 CPU 异常或中断仿真波形图 (7)

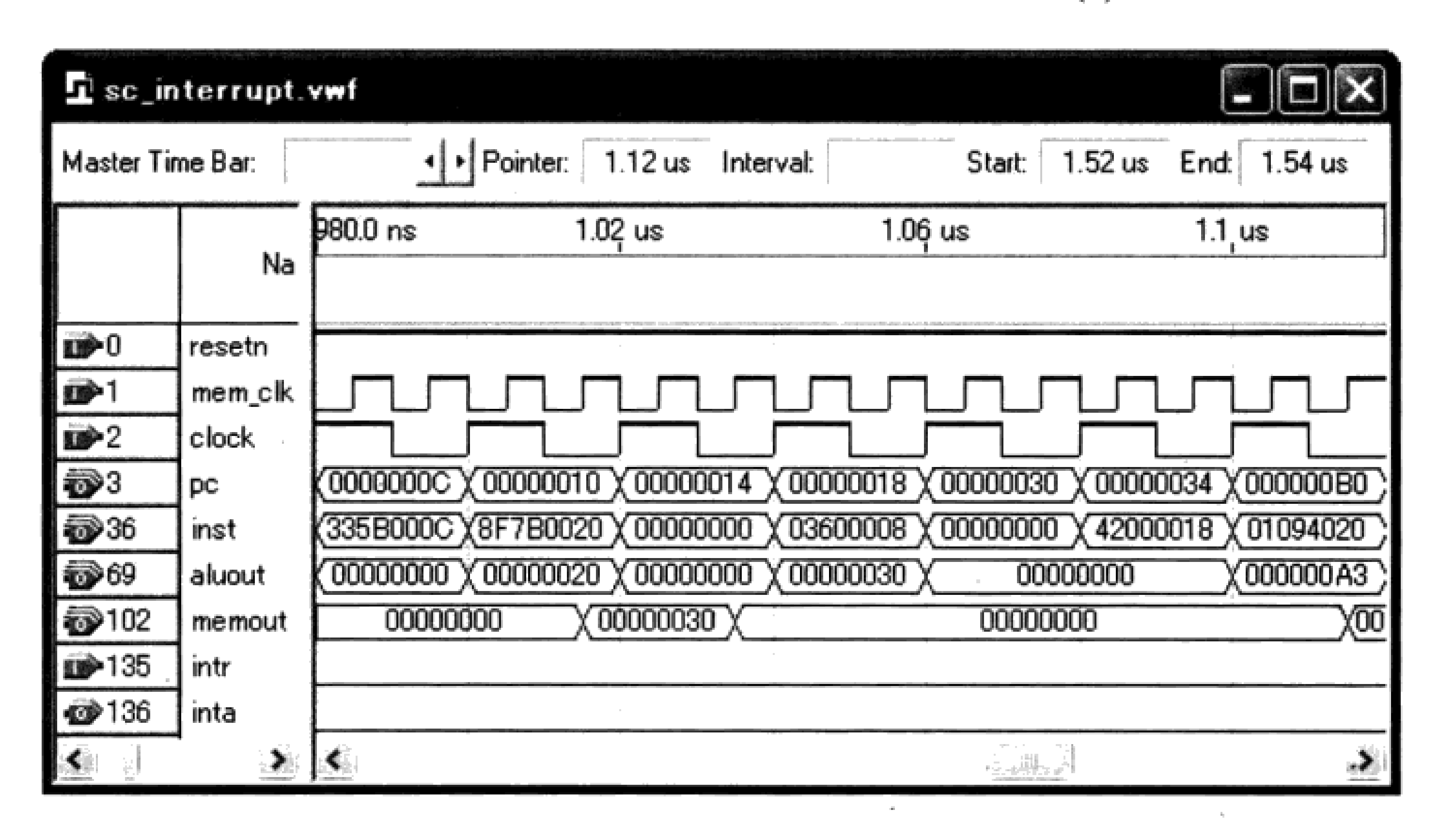


图 6.19　单周期 CPU 异常或中断仿真波形图 (8)

时候。CPU 执行完该指令后，转去 0x00000008。它是执行以下 7 条指令的波形。

```
26: 00000000; % (98)             nop                    %
27: 34040050; % (9c) Int:        ori  r4, r1, 0x50      %
28: 20050004; % (a0)             addi r5, r0,  4        %
29: 00004020; % (a4)             add  r8, r0, r0        %
2a: 8c890000; % (a8) loop:       lw   r9, 0(r4)         %
2b: 20840004; % (ac)             addi r4, r4,  4        %
 2: 401a6800; % (08) EXC_BASE:   mfc0 r26, C0_CAUSE     %
```

图 6.19 演示从中断处理程序返回到 PC = 0x000000B0，执行以下 7 条指令。

```
 3: 335b000c; % (0c)             andi r27, r26, 0xc     %
 4: 8f7b0020; % (10)             lw   r27, j_table(r27) %
 5: 00000000; % (14)             nop                    %
 6: 03600008; % (18)             jr   r27               %
 c: 00000000; % (30) int_entry:  nop                    %
 d: 42000018; % (34)             eret                   %
2c: 01094020; % (b0)             add  r8, r8, r9        %
```

程序还没有结束，但我们的直播就到此为止。

6.4 习题

1. 用 Verilog HDL 设计图 6.7 的中断控制器电路。
2. 用 Verilog HDL 设计图 6.8 的 Daisy-Chain 电路。
3. 在图 6.11 的基础上设计带有多个中断请求信号的 CPU。
4. 利用对未实现的指令异常的处理，我们可以用软件的方法实现那些硬件不支持的指令。如果在未实现的指令的异常处理程序中能够通过 EPC 来读取引起异常的指令，我们便可以用现有的指令来仿真。读取引起异常的指令的方法如下。

```
uni_entry:                      # Unimpl. inst. exception handler
           mfc0  r26, C0_EPC # get EPC
           lw    r26, 0(r26) # get the unimpl. instruction
emulator:  ...               # implement it here
finish:    mfc0  r26, C0_EPC # get EPC
           addi  r26, r26, 4 # EPC + 4
           mtc0  r26, C0_EPC # EPC <-- EPC + 4
           eret              # return from exception
```

   但遗憾的是本章的 CPU 连接了两个分开的存储器模块：数据存储器和指令存储器。用 lw 指令不能从指令存储器中取数据。想想有没有什么招数实现对硬件不支持的指令的软件仿真。
5. 试书写测试程序来实现中断嵌套 (注意要保存 EPC)。

第 7 章　多周期 CPU 及其 Verilog HDL 设计

我们已经在第 5 章讲述了单周期 CPU 的设计方法。单周期 CPU 用一个时钟周期执行一条指令。而确定时钟周期的时间长度时要考虑执行时间最长的指令，以此定出 CPU 的时钟频率。我们知道，一旦时钟频率确定之后，一个周期的时间长度也就固定了。因此不管每条指令的复杂程度如何，单周期 CPU 都花费相同的时间去执行每条指令，这就造成了时间上的浪费，因为简单的指令根本不需要那么长的时间。

本章讨论多周期 CPU 的设计方法并给出 Verilog HDL 代码。多周期 CPU 的中心思想是把一条指令的执行分成若干个小周期，根据每条指令的复杂程度，使用不同数量的小周期去执行。许多个小周期加在一起相当于单周期 CPU 中的一个周期。在不会引起与单周期混淆的情况下，我们简称小周期为周期。

7.1　把一条指令的执行分成若干个周期

在我们实现的 20 条指令中，最复杂的指令就是 lw rt, offset (rs) 了。它需要 5 个周期，其整个执行过程为：

1) 根据 PC 取指令，并把 PC 加 4；
2) 对指令译码并读出 rs 寄存器的内容；
3) 计算存储器地址：由 rs 寄存器的内容与指令中的偏移量 offset 相加得到；
4) 使用计算好的地址访问存储器，从中读出一个 32 位的数据；
5) 最后把该数据写入寄存器堆中的 rt 寄存器。

而最简单的指令非 j address 莫属了，两个周期就行：

1) 根据 PC 取指令，并把 PC 加 4；
2) 指令中的 address 左移两位与 PC (加过 4 了) 的高 4 位拼接起来，写入 PC。

ALU 计算类型的指令需要 4 个周期：

1) 根据 PC 取指令，并把 PC 加 4；
2) 读出 rs 和 rt 两个寄存器的内容；
3) 由 ALU 完成对两个寄存器数据 (或一个立即数) 的计算；
4) 最后把计算结果写入寄存器堆中的 rd (或 rt) 寄存器。

转移类型的指令，例如 beq rs, rt, offset，需要 3 个周期：

1) 根据 PC 取指令，并把 PC 加 4；
2) 读出 rs 和 rt 两个寄存器的数据并锁存，同时 ALU 计算转移地址并锁存；
3) 由 ALU 比较两个寄存器数据，并决定是否把转移地址写入 PC。

多周期 CPU 与单周期 CPU 的时序比较见图 7.1。表 7.1 列出了每条指令所用的周期数。注意，多周期 CPU 也忽略了 MIPS 转移类指令的延迟转移特性，我们将在流水线 CPU 中实现延迟转移。以下我们讨论在每个周期执行指令时所需的电路。

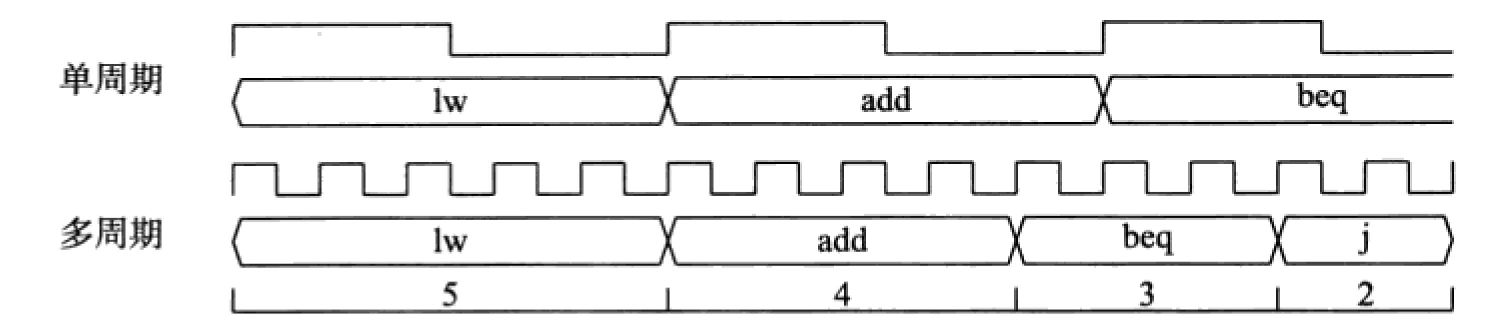


图 7.1 多周期 CPU 与单周期 CPU 的时序比较

表 7.1 每条指令所用的周期数

指 令	意 义	周期数
`add rd, rs, rt`	寄存器加	4
`sub rd, rs, rt`	寄存器减	4
`and rd, rs, rt`	寄存器与	4
`or rd, rs, rt`	寄存器或	4
`xor rd, rs, rt`	寄存器异或	4
`sll rd, rt, sa`	左移	4
`srl rd, rt, sa`	逻辑右移	4
`sra rd, rt, sa`	算术右移	4
`jr rs`	寄存器跳转	2
`addi rt, rs, immediate`	立即数加	4
`andi rt, rs, immediate`	立即数与	4
`ori rt, rs, immediate`	立即数或	4
`xori rt, rs, immediate`	立即数异或	4
`lw rt, offset(rs)`	取字	5
`sw rt, offset(rs)`	存字	4
`beq rs, rt, offset`	相等转移	3
`bne rs, rt, offset`	不等转移	3
`lui rt, immediate`	设置高位	4
`j address`	跳转	2
`jal address`	调用	2

7.1.1 取指令周期 IF

取指令周期 IF (Instruction Fetch) 做两件事情：取指令和 PC + 4。电路见图 7.2。

```
IR <-- Memory[PC];
PC <-- PC + 4;
```

多周期 CPU 设计的基本原则是在每个周期结束时把本周期的结果保存在某个地方以便下一个周期使用。例如在图 7.2 的电路中我们设置了一个带有写使能端的指令寄存器 IR (Instruction Register)。这样，即使 PC 的值改变了，只要 IR 的写使能端 wir (Write IR) 无效，IR 中的指令就不会改变。

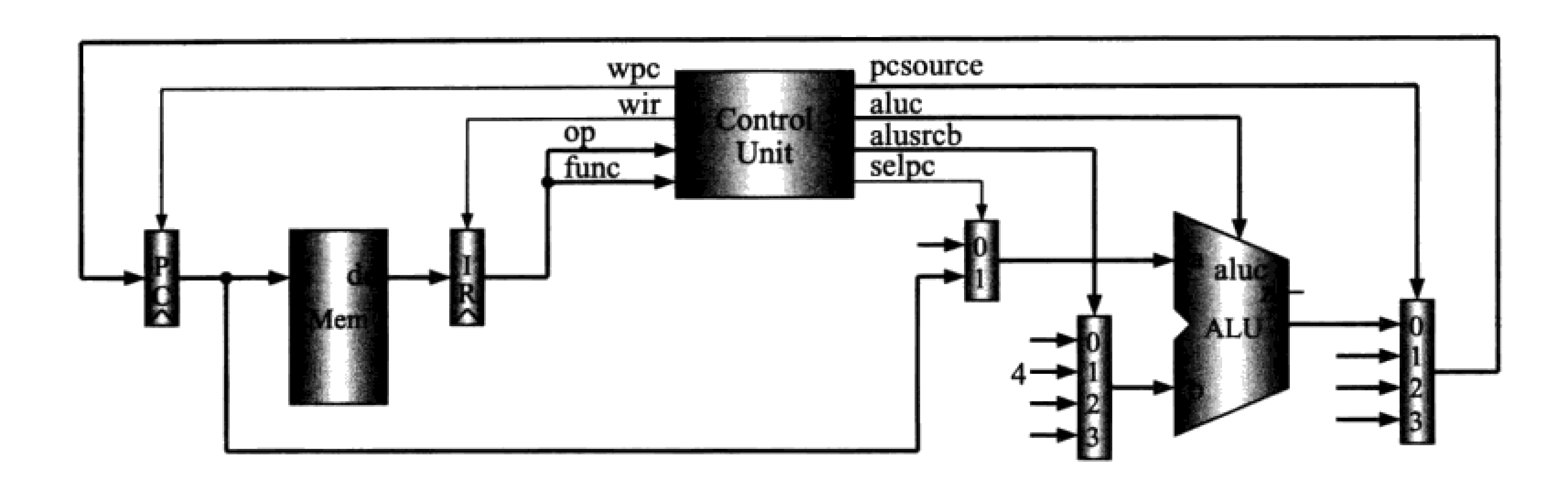


图 7.2　取指令周期 IF

由于该周期 ALU 无事可做，我们可以把 PC + 4 的工作分派给它，这样就没必要再使用一个专门的加法器了。由于不是每个周期 PC 都加 4，所以也给 PC 加了一个写使能 wpc (Write PC)。我们还专门在 ALU 前面加了两个多路器，它们的选择信号分别为 selpc (Select PC) 和 alusrcb (ALU Source B)。该周期的控制信号的输出应该为：wir = 1 (写 IR)，wpc = 1 (写 PC)，selpc = 1 (选择 PC)，alusrcb = 1 (选择 4)，aluc = x000 (ALU 做加法)，pcsource = 0 (选择 PC + 4)。

7.1.2　指令译码周期 ID

除了 j、jal 和 jr 三条指令之外，其他指令在指令译码 (Instruction Decode) 周期做以下三件事情：

```
A <-- RegisterFile[rs];
B <-- RegisterFile[rt];
C <-- PC + sign_extend(offset) << 2;
```

即根据寄存器号 rs 和 rt 从寄存器堆读出两个 32 位数据，将它们分别存放在寄存器 A 和 B 中。与此同时，ALU 计算转移地址，即把 PC (在 IF 周期已经加过 4 了) 与指令中的偏移量左移两位相加。偏移量要进行符号扩展。计算出的转移地址写入寄存器 C。这项工作完全是为了转移指令 (beq 和 bne) 而做的，其他指令并不需要它。这部分的电路如图 7.3 所示。

控制信号的输出应该为：wir = 0 (不写 IR)，wpc = 0 (不写 PC)，selpc = 1 (选择 PC)，alusrcb = 3 (选择左移两位后的 offset)，sext = 1 (符号扩展)，aluc = x000 (ALU 做加法)，pcsource = x (任意，因为不写 PC)。注意寄存器 A、B 和 C 没有写使能端，这意味着每个时钟周期都无条件地把数据写入其中。

指令 j、jal 和 jr 在本周期将完成最后的操作，即无条件跳转。jr 指令从 rs 寄存器中得到转移地址；j 和 jal 指令得到转移地址的方法是把 PC 的高 4 位与指令中的 address 左移两位拼接。另外，jal 指令还要把 PC 的值写入 31 号寄存器 r31。

```
j:   PC <-- {PC[31:28],address,00};
```

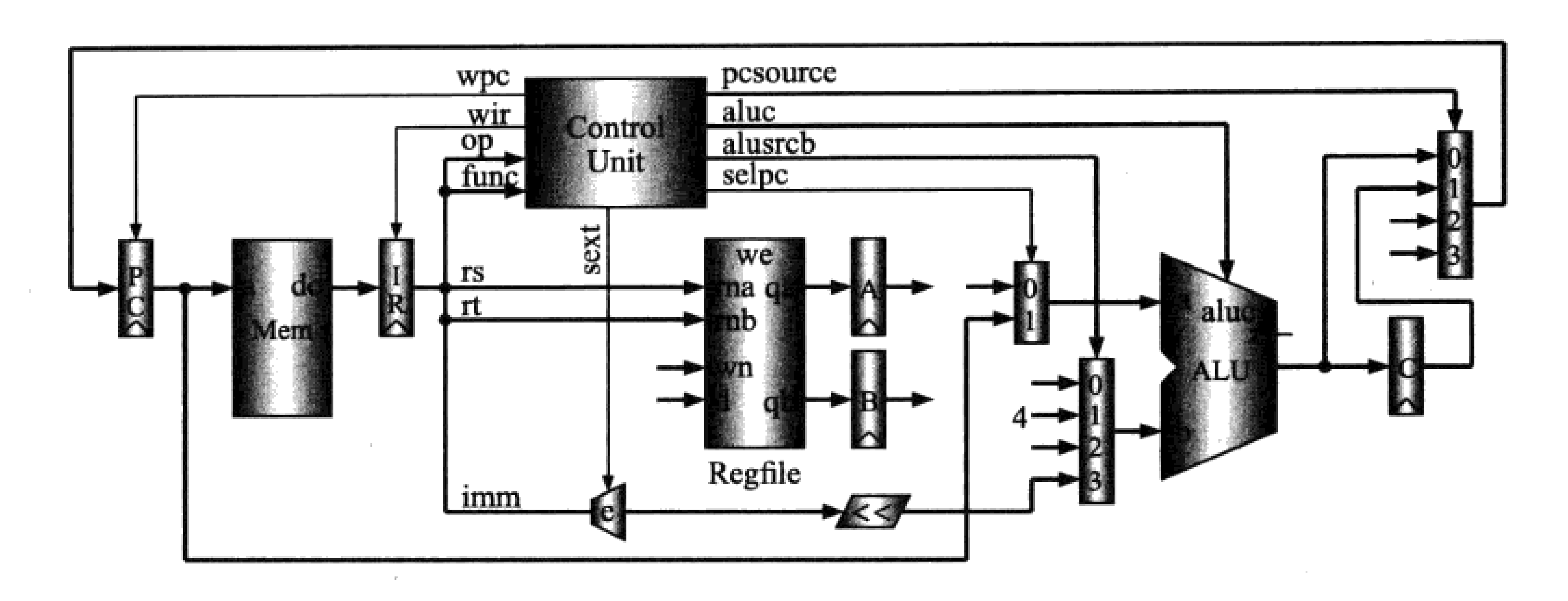


图 7.3 指令译码周期 ID (不包括 j、jal 和 jr 指令)

```
jal: RegisterFile[31] <-- PC;
     PC <-- {PC[31:28],address,00};
jr:  PC <-- RegisterFile[rs];
```

实现这三条指令的电路如图 7.4 所示。我们把图 7.3 的电路也加进去了。由于这三条指令都是无条件转移，所以 wpc = 1。当指令为 jr 时，pcsource = 2 (选择 rs 寄存器的内容)；当指令为 j 或 jal 时，pcsource = 3；如果是 jal 指令，jal = 1 (生成目的寄存器号 31、选择 PC 送往寄存器堆的 d 端)，wreg = 1 (保存返回地址)。

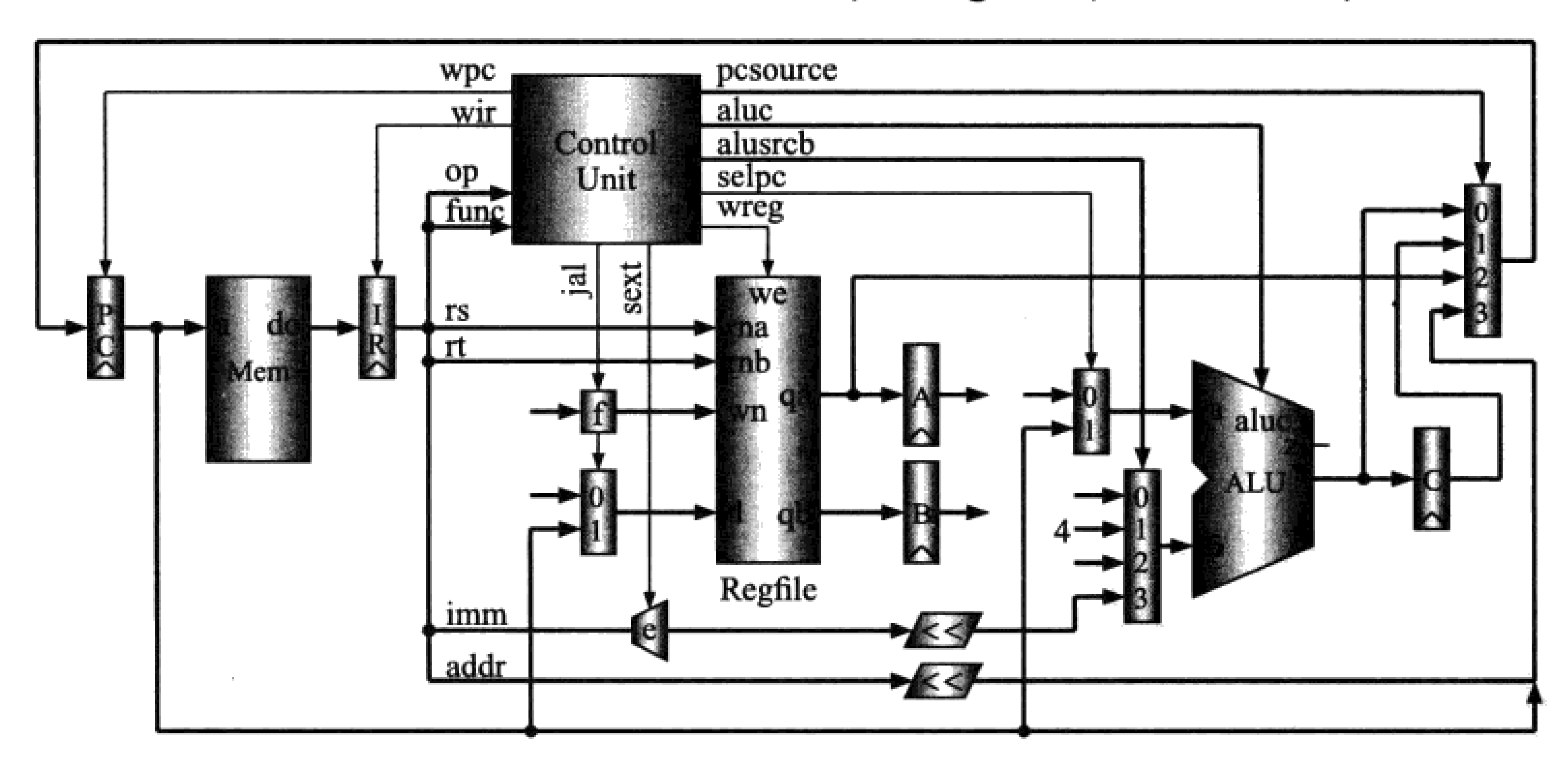


图 7.4 指令译码周期 ID (包括 j、jal 和 jr 指令)

7.1.3 指令执行周期 EXE

三条指令 j、jal 和 jr 在 ID 周期已经完成了它们的使命，而其他指令则进入执行

(Execution) 周期。这个周期要分很多种情况加以讨论，因为不同类型的指令在执行时有不同的电路要求。首先看条件转移指令 beq 和 bne 的执行情况：

```
beq: if (A == B) PC <-- C;
bne: if (A != B) PC <-- C;
```

其中的 (A == B) 和 (A != B) 是判断 A、B 两个寄存器的内容是否相等。注意寄存器 C 的内容，它是 ID 周期计算出的转移地址。这两条指令在本周期结束操作。图 7.5 新加的部分是执行这两条指令所需的电路，即判断是否相等由 ALU 完成。判断结果由 z 送出，控制部件使用它来产生 wpc 信号：如果条件满足，修改 PC，即 wpc = beq z + bne $\bar{z}$。这时的 pcsource = 1 (选择寄存器 C 的内容)。条件转移指令在本周期结束。

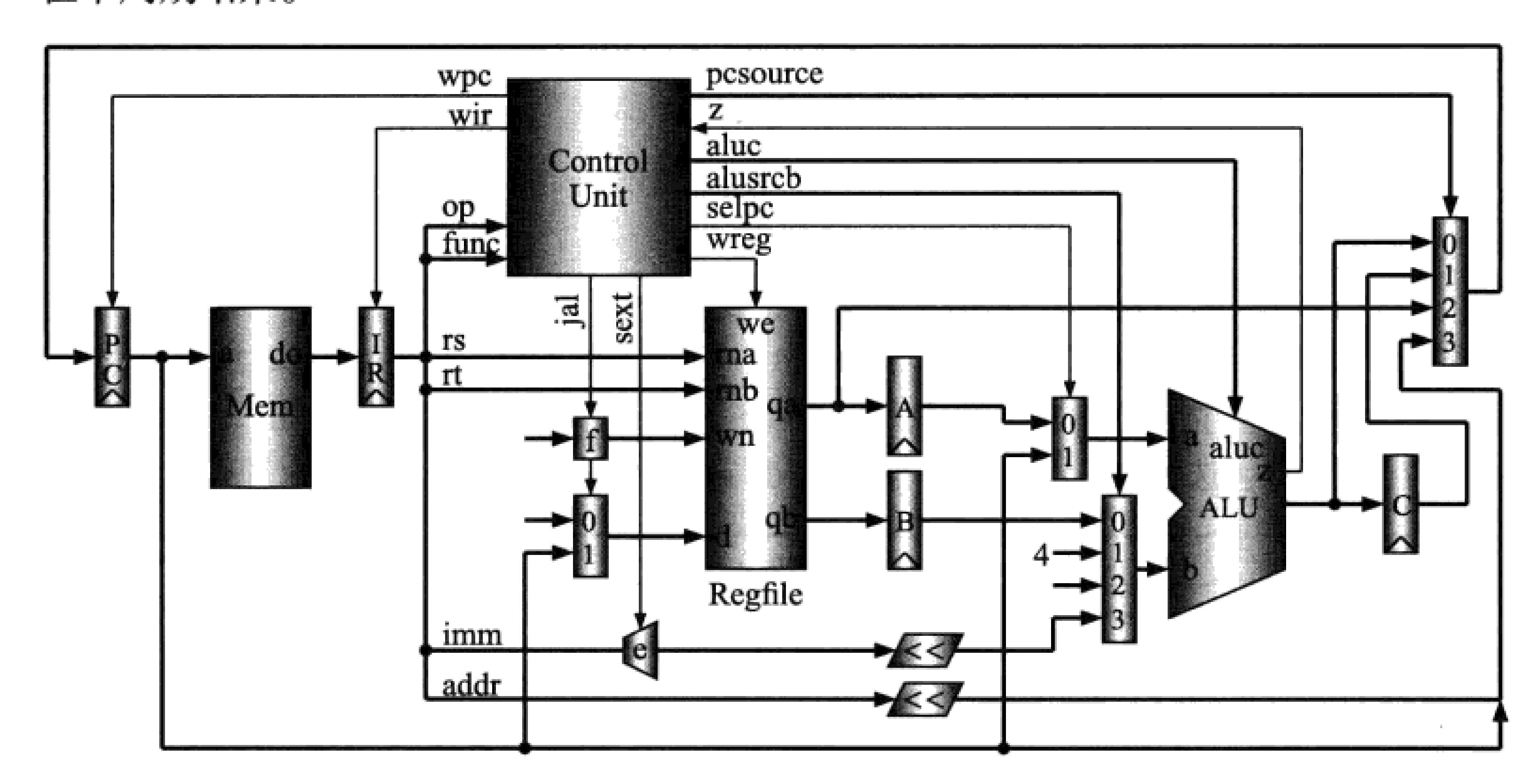


图 7.5 转移指令和寄存器类型的算术和逻辑运算指令的执行周期 EXE

寄存器类型的算术和逻辑运算指令 (add、sub、and、or 和 xor) 所需电路与转移指令相同 (见图 7.5)，只是控制信号不同。这些指令完成如下的操作：

```
add/sub/and/or/xor: C <-- A op B;
```

立即数类型的指令 (addi、andi、ori、xori、lw、sw 和 lui) 所需电路如图 7.6 所示，扩展后的立即数连接到了 ALU 左边的四选一多路器的输入端。这些指令完成如下的操作：

```
addi:          C <-- A +  sign_extend(immediate);
andi/ori/xori: C <-- A op zero_extend(immediate);
lw/sw:         C <-- A +  sign_extend(offset);
lui:           C <-- immediate << 16;
```

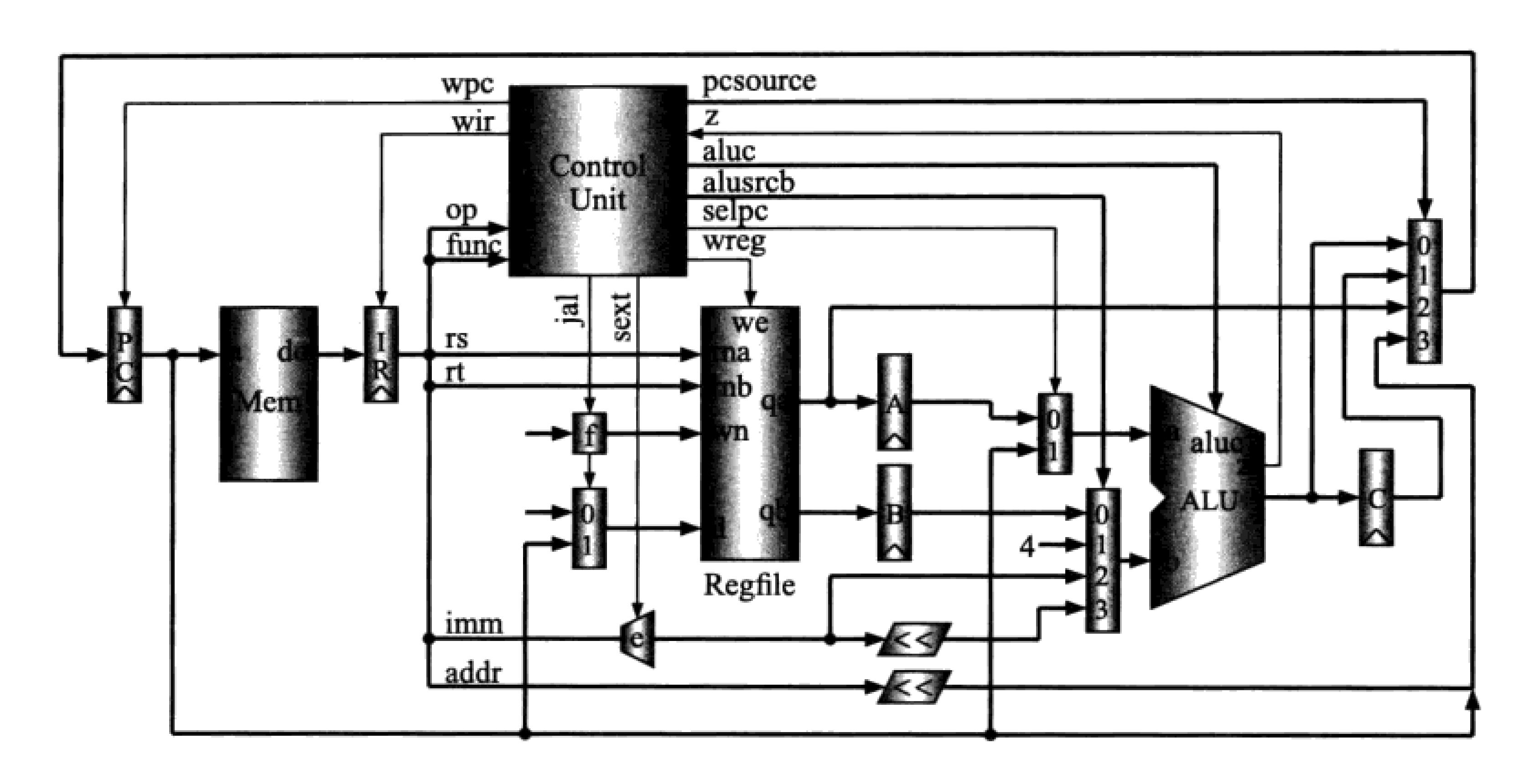


图 7.6 立即数类型指令的执行周期 EXE

由于 immediate 和 offset 都是指令中的 16 位立即数，我们在图中用 imm 表示。执行这些指令时，控制信号 alusrcb = 2 (选择立即数)。

移位指令 (sll、srl 和 sra) 所需电路如图 7.7 所示。我们又用了一个二选一多路器，选择信号为 shift，为 1 时选择指令中的 sa。这些指令完成如下的操作：

```
sll: C <-- B << sa;
srl: C <-- B >> sa;
sra: C <-- signed(B) >>> sa;
```

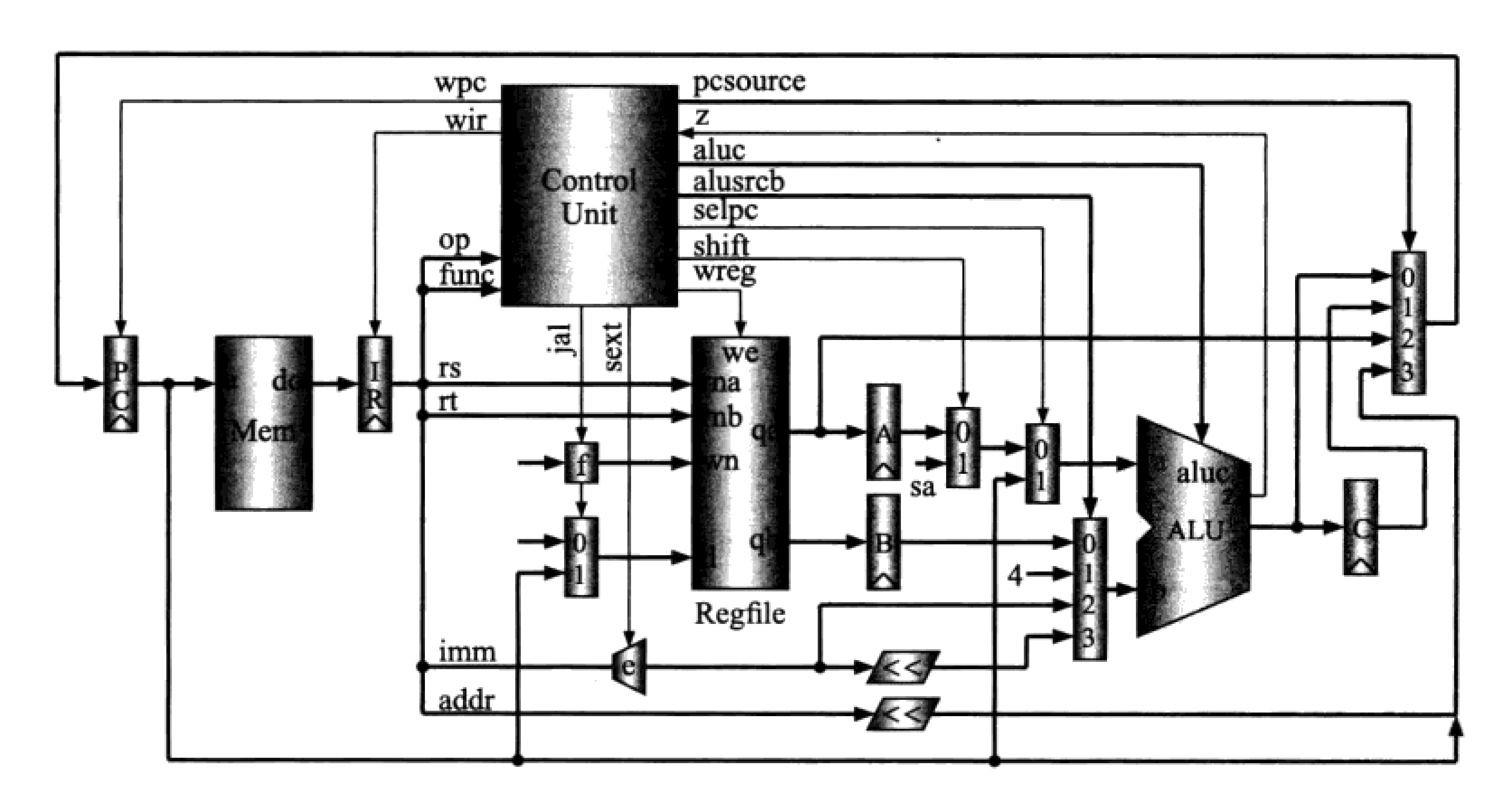


图 7.7 移位类型指令的执行周期 EXE

注意，指令中的 sa 只有 5 位，送往二选一多路器时要扩展成 32 位。ALU 做移位操作时只使用 a 输入端的最低 5 位，因此扩展时的高 27 位可以设置成任意值。另外，C <-- signed (B) >>> sa 意味着把 B 算术右移 sa 位，在 Verilog HDL 中，可以使用 $signed (B) >>> sa 语句。

本周期结束时，lw 和 sw 指令将进入存储器访问周期 MEM；其他指令将进入结果写回周期 WB。

7.1.4　存储器访问周期 MEM

只有 lw 和 sw 指令进入存储器访问周期 MEM (Memory Access)。在 EXE 周期，我们已经计算出了存储器地址，存放在寄存器 C 中。lw 指令从存储器中取数据；sw 指令往存储器中存数据：

```
lw: DR <-- Memory[C];
sw: Memory[C] <-- B;
```

这两条指令在存储器访问周期所需的硬件电路如图 7.8 所示。我们可以像单周期计算机那样，设置单独的数据存储器。但我们这里把指令存储器和数据存储器合二为一，只用一个存储器模块。因为取指令时使用 PC 作为存储器地址，而读取存储器数据时使用 ALU 计算出的地址，所以我们在存储器的地址输入端使用了一个二选一多路器。二选一多路器的选择信号为 iord。执行 lw 或 sw 指令时，iord = 1，选择寄存器 C 中的地址。我们还使用了一个新的寄存器：DR (Data Register)。从存储器取出的数据存放在 DR 中。

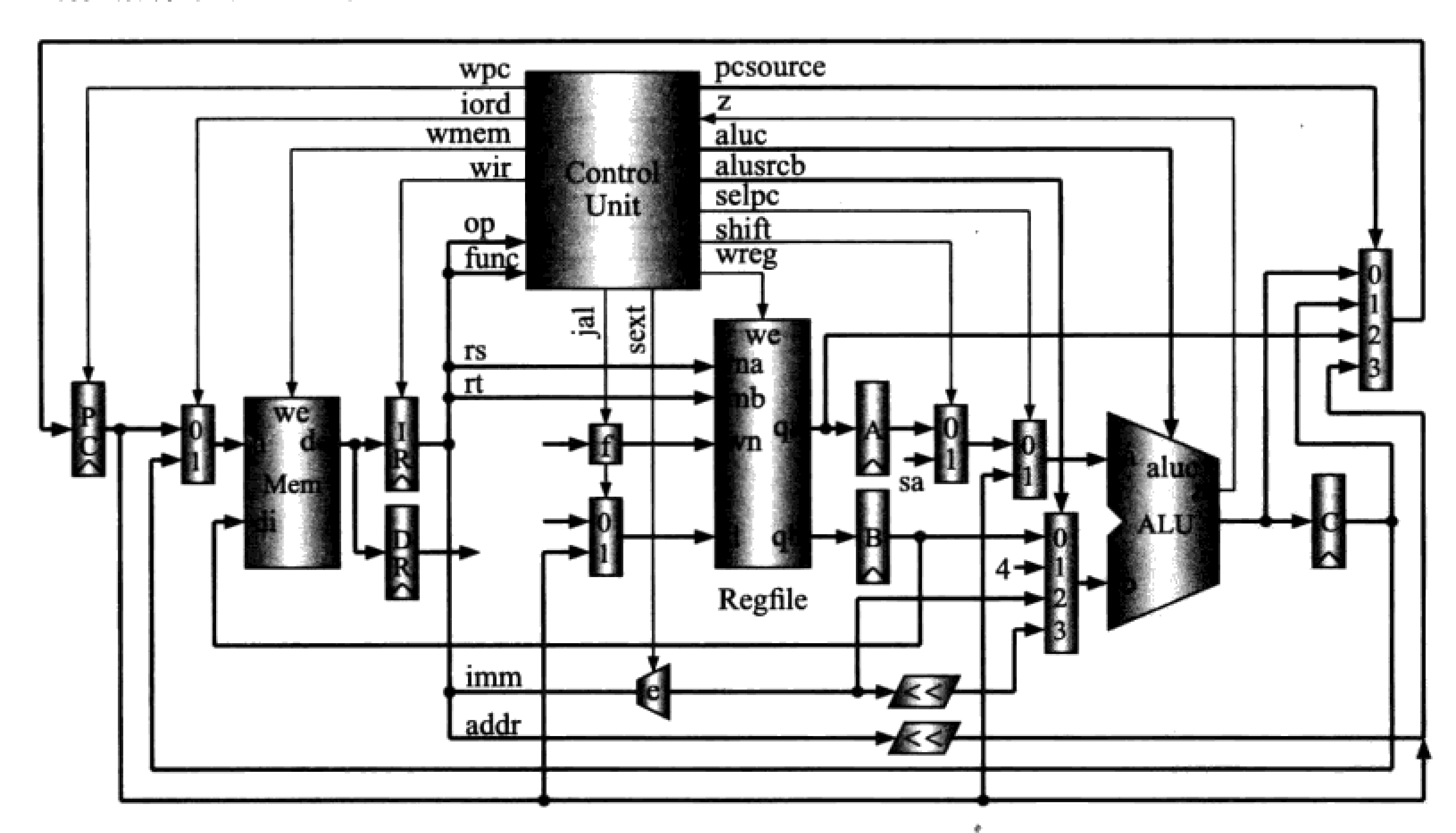


图 7.8　存储器访问周期 MEM

执行 sw 指令时，把寄存器 B 中的内容写入存储器。由于寄存器 B 没有写使能端，即每个周期都更新寄存器 B，所以我们要保证在 EXE 周期向寄存器 B 中写入的是我们所希望的数据，即：B ← RegisterFile[rt]。因为在 ID 周期结束时我们并没有向 IR 寄存器写入新的指令，所以在 EXE 周期向寄存器 B 写入的仍然是 RegisterFile[rt]。sw 指令在本周期结束，lw 指令将进入 WB 周期。注意图 7.8 包含了除 WB 之外的所有电路。

7.1.5 结果写回周期 WB

结果写回周期 WB (Write Back) 把 ALU 的计算结果或者从存储器取来的数据写入寄存器堆。目的寄存器号有 rd 和 rt 之分:

```
add/sub/and/or/xor/sll/srl/sra: RegisterFile[rd] <-- C;
addi/andi/ori/xori/lui:         RegisterFile[rt] <-- C;
lw:                             RegisterFile[rt] <-- DR;
```

结果写回周期 WB 的电路图见图 7.9。控制信号 m2reg 选择 DR 或 C；regrt 选择 rd 或 rt；jal = 0；wreg = 1。

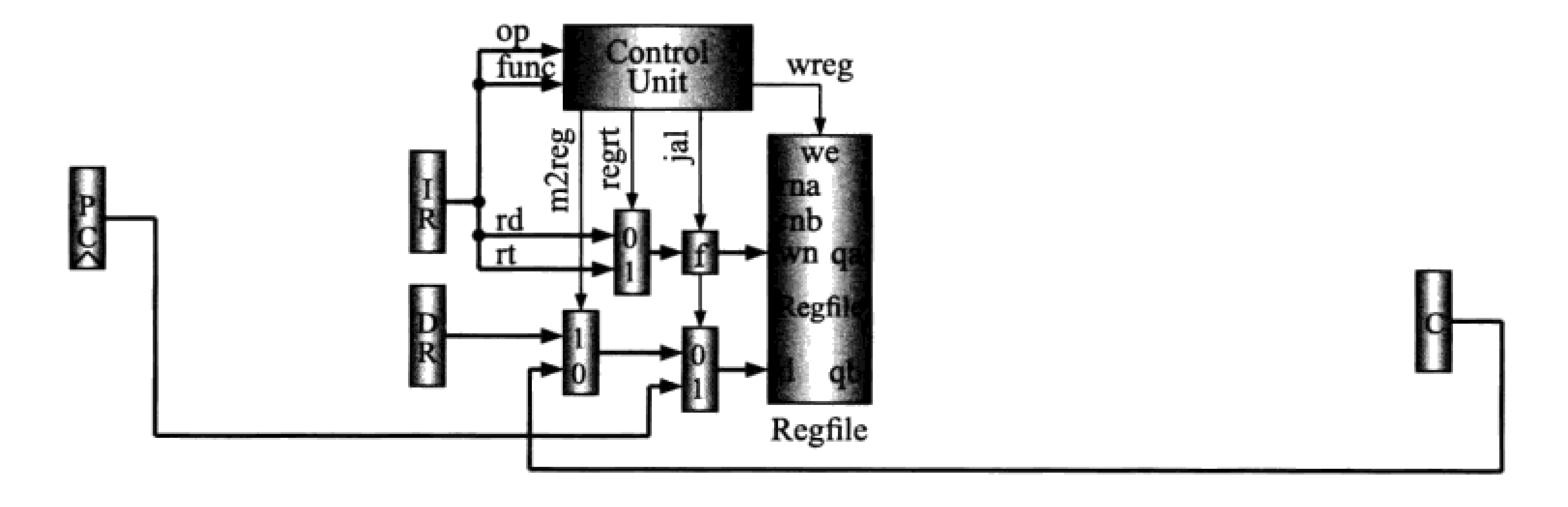


图 7.9 结果写回周期 WB (忽略了其他部分的电路)

7.2 多周期 CPU 的总体电路及 Verilog HDL 代码

7.2.1 多周期 CPU 的总体电路

综合 7.1 节的描述，我们得到如图 7.10 所示的多周期 CPU 加上存储器的总体电路图。除了控制部件，其他所有的部件都已经在设计单周期 CPU 时描述过了。

7.2.2 多周期 CPU 的 Verilog HDL 代码

以下的模块 mccomp 是多周期 CPU 加上存储器的 Verilog HDL 代码。它调用多周期 CPU 模块 mccpu 和存储器模块 mcmem。我们将在“存储器及测试程序设计”一节介绍存储器 mcmem 模块。

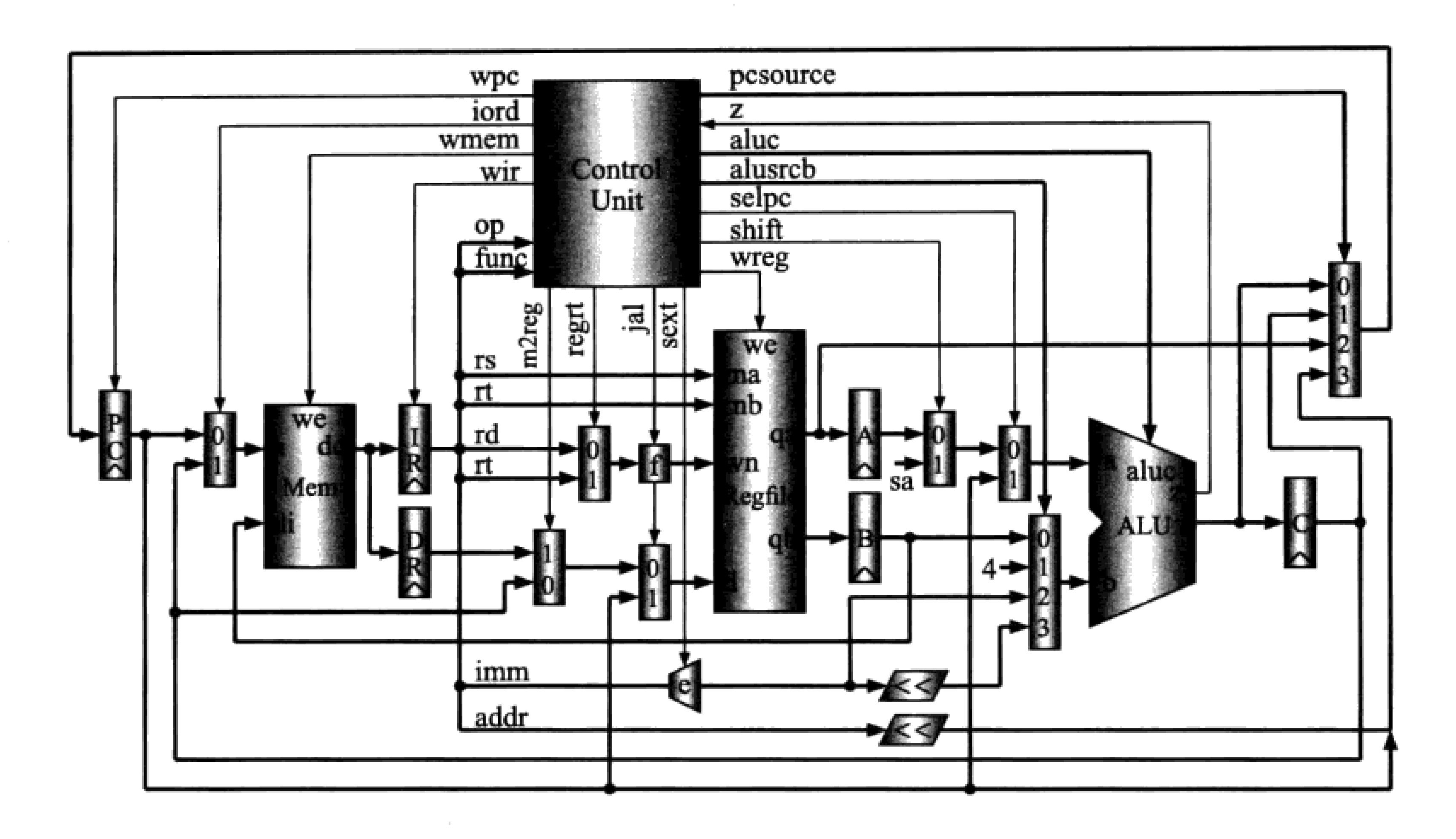


图 7.10 多周期 CPU + 存储器的总体电路图

```
module mccomp (clock,resetn,q,a,b,alu,adr,tom,fromm,pc,ir,mem_clk);
    input clock,resetn,mem_clk;
    output [31:0] a,b,alu,adr,tom,fromm,pc,ir;
    output [2:0]  q;
    wire          wmem;
    mccpu mc_cpu (clock,resetn,fromm,pc,ir,a,b,alu,wmem,adr,tom,q);
    mcmem memory (clock,fromm,tom,adr,wmem,mem_clk,mem_clk);
endmodule
```

以下的模块 mccpu 是结构描述风格的多周期 CPU 的 Verilog HDL 代码，它调用的控制部件 mccu 模块在下一节给出。其他被调用的模块应该已经全部介绍过了。

```
module mccpu (clock,resetn,frommem,pc,inst,alua,alub,alu,wmem,madr,tomem,
              state);
    input [31:0] frommem;
    input        clock,resetn;
    output [31:0] pc,inst,alua,alub,alu,madr,tomem;
    output [2:0] state;
    output       wmem;
    wire [3:0]   aluc;
    wire [4:0]   reg_dest;
    wire         z,wpc,wir,wmem,wreg,iord,regrt,m2reg,shift,selpc,jal,sext;
    wire [31:0] npc,rega,regb,regc,mem,qa,qb,res,opa,bra,alub,alu_mem;
    wire [1:0]   alusrcb, pcsource;
    wire [31:0] sa = {27'b0,inst[10:6]};
    mccu control_unit (inst[31:26],inst[5:0],z,clock,resetn,
```

```
                        wpc, wir, wmem, wreg, iord, regrt, m2reg, aluc,
                        shift,selpc, alusrcb, pcsource, jal, sext, state);
    wire       e = sext & inst[15];
    wire [15:0] imm = {16{e}};
    wire [31:0] immediate = {imm,inst[15:0]};
    wire [31:0] offset = {imm[13:0],inst[15:0],1'b0,1'b0};
    dffe32 ip (npc,clock,resetn,wpc,pc);
    dffe32 ir (frommem,clock,resetn,wir,inst);
    dff32  dr (frommem,clock,resetn,mem);
    dff32  ra (qa,clock,resetn,rega);
    dff32  rb (qb,clock,resetn,regb);
    dff32  rc (alu,clock,resetn,regc);
    assign tomem = regb;
    mux2x5 reg_wn (inst[15:11],inst[20:16],regrt,reg_dest);
    wire [4:0] wn = reg_dest | {5{jal}}; // jal: r31 <-- p4;
    mux2x32 mem_address (pc,regc,iord,madr);
    mux2x32 result (regc,mem,m2reg,alu_mem);
    mux2x32 link   (alu_mem,pc,jal,res);
    mux2x32 oprand_a (rega,sa,shift,opa);
    mux2x32 alu_a (opa,pc,selpc,alua);
    mux4x32 alu_b (regb,32'h4,immediate,offset,alusrcb,alub);
    mux4x32 nextpc (alu,regc,qa,jpc,pcsource,npc); // next pc
    regfile rf (inst[25:21],inst[20:16],res,wn,wreg,clock,resetn,qa,qb);
    wire [31:0] jpc = {pc[31:28],inst[25:0],1'b0,1'b0};
    alu alunit (alua,alub,aluc,alu,z);
endmodule
```

7.3 用有限状态机实现多周期 CPU 的控制部件

我们可以用典型的时序电路来实现多周期 CPU 的控制部件。重要的工作是确定状态转移图。状态转移图不是唯一的，只要能实现 7.1 节描述的各条指令所经过的周期 (状态) 即可。

7.3.1 多周期 CPU 的控制部件的状态转移图

本小节给出的只是一种可能的状态转移图，见图 7.11。我们这里使用了最少的状态数。从图中可以看出，三条跳转指令用两个周期；两条条件转移指令用三个周期；lw 指令用五个周期；其余指令均用四个周期。图中的五个状态分别有五个名字，而且我们为每个状态分别指定了一个唯一的 3 位二进制数。实际上，这 3 位二进制数是每个状态的“身份证号码”，不允许两个不同的状态有相同的号码。

7.3.2 多周期 CPU 的控制部件的总体结构

图 7.12 所示的是多周期 CPU 控制部件的电路结构图。这是一个非常典型的时序

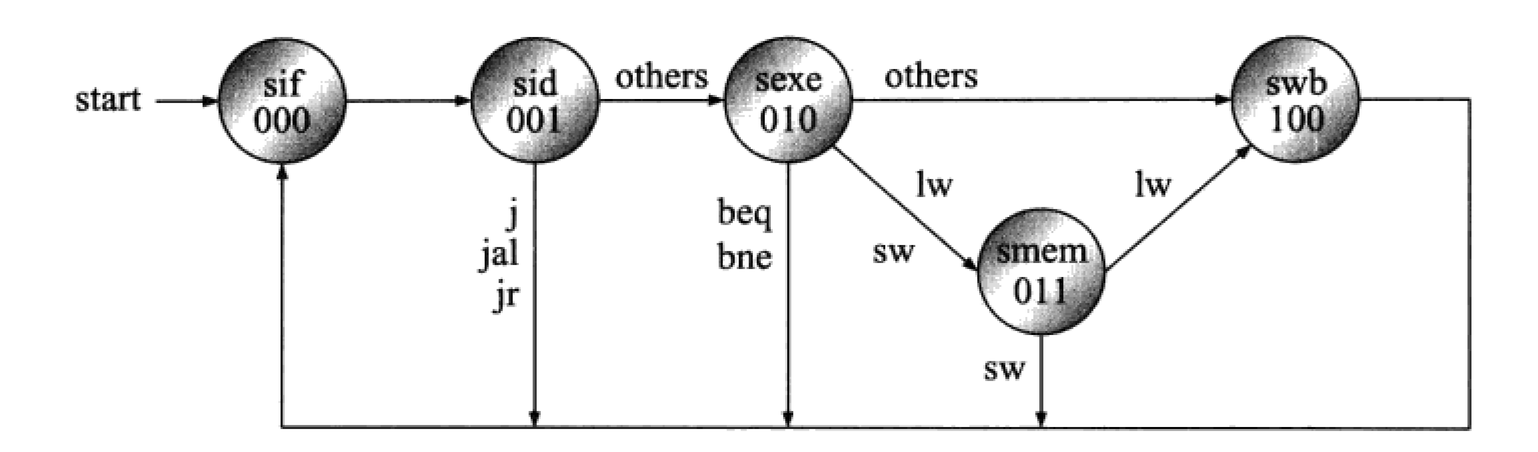


图 7.11 多周期 CPU 控制部件的状态转移图

电路。D 触发器保存 (指出) 当前状态，其余两个模块是组合电路，分别产生表示下一状态的 3 位二进制数和控制信号。

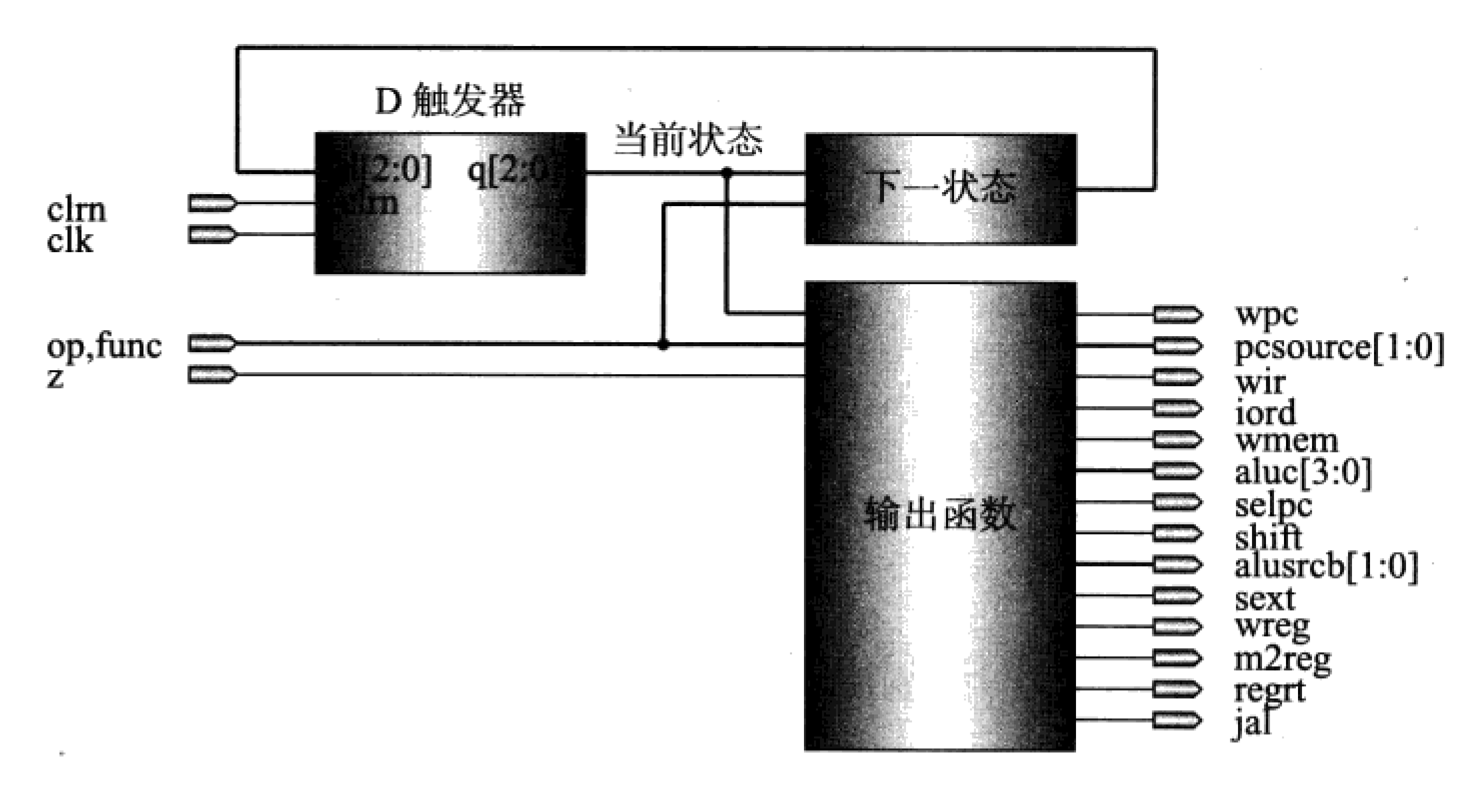


图 7.12 多周期 CPU 控制部件的电路结构图

7.3.3 下一状态函数

图 7.12 中的“下一状态”模块是组合电路，用于产生下一状态的信息，我们用 d[2:0] 表示它。d[2:0] 在时钟的上升沿处被存入 D 触发器，D 触发器的输出 q[2:0] 是当前状态。根据图 7.11 所示的控制部件的状态转移图，我们有如表 7.2 所示的真值表。我们的目的是求出 d[2:0] 每一位的逻辑表达式。

由真值表，我们得到如下的逻辑表达式，其中的 sif、sid、sexe、smem 和 swb 表示状态，可以认为是中间变量。

$$\text{sif} = \overline{q[2]}\ \overline{q[1]}\ \overline{q[0]};$$
$$\text{sid} = \overline{q[2]}\ \overline{q[1]}\ q[0];$$
$$\text{sexe} = \overline{q[2]}\ q[1]\ \overline{q[0]};$$

表 7.2　下一状态函数的真值表

当前状态		输　入	下一状态	
状态	q[2 : 0]	op[5 : 0]	状态	d[2 : 0]
sif	0 0 0	x	sid	0 0 1
sid	0 0 1	i_j	sif	0 0 0
		i_jal	sif	0 0 0
		i_jr	sif	0 0 0
		others	sexe	0 1 0
sexe	0 1 0	i_beq	sif	0 0 0
		i_bne	sif	0 0 0
		i_lw	smem	0 1 1
		i_sw	smem	0 1 1
		others	swb	1 0 0
smem	0 1 1	i_lw	swb	1 0 0
		i_sw	sif	0 0 0
swb	1 0 0	x	sif	0 0 0

$$\mathrm{smem} = \overline{q[2]}\ q[1]\ q[0];$$
$$\mathrm{swb} = q[2]\ \overline{q[1]}\ \overline{q[0]};$$
$$d[0] = \mathrm{i_sif} + \mathrm{i_sexe}\ (\mathrm{i_lw} + \mathrm{i_sw});$$
$$d[1] = \mathrm{sid}\ \overline{(\mathrm{i_j} + \mathrm{i_jal} + \mathrm{i_jr})} + \mathrm{sexe}\ (\mathrm{i_lw} + \mathrm{i_sw});$$
$$d[2] = \mathrm{sexe}\ \overline{(\mathrm{i_beq} + \mathrm{i_bne} + \mathrm{i_lw} + \mathrm{i_sw})} + \mathrm{smem}\ \mathrm{i_lw};$$

表达式中的指令译码与单周期 CPU 相同。有了逻辑表达式，我们可以画出逻辑图，此处就不再给出了。

7.3.4　控制信号的产生

控制部件中的另一部分“输出函数”也是组合电路，用来产生控制信号。表 7.3 是控制信号的真值表。表中只列出了在 sexe 状态下执行 add 指令时各控制信号的取值以及控制信号 wmem 的所有取值。我们有：

$$\mathrm{wmem} = \mathrm{smem}\ \ \mathrm{i_sw};$$

作为练习题，试把表 7.3 填满，并用逻辑图输入的方法设计多周期 CPU。

7.3.5　控制部件的 Verilog HDL 代码

以下是控制部件的 Verilog HDL 代码。注意，代码中并没有使用前两小节给出的逻辑表达式，而是根据状态及指令直接对控制信号赋值。值得一提的是实现状态转移的方法：我们使用了中间变量 next_state，意为下一状态。在当前状态中，根据指令对 next_state 赋值，并在每个时钟上升沿把 next_state 打入状态寄存器。这也是用 Verilog HDL 实现有限状态机时常用的方法。

表7.3　控制信号的真值表

输入			输出													
状态	指令	z	wpc	pcsource[1:0]	wir	iord	wmem	aluc[3:0]	selpc	shift	alusrcb[1:0]	sext	wreg	m2reg	regrt	jal
sif	x	x					0									
sid	i_j	x					0									
	i_jal	x					0									
	i_jr	x					0									
	others	x					0									
sexe	i_add	x	0	x x	0	x	0	x 0 0 0	0	0	0 0	x	0	x	x	0
	i_sub	x					0									
	i_and	x					0									
	i_or	x					0									
	i_xor	x					0									
	i_sll	x					0									
	i_srl	x					0									
	i_sra	x					0									
	i_addi	x					0									
	i_andi	x					0									
	i_ori	x					0									
	i_xori	x					0									
	i_lw	x					0									
	i_sw	x					0									
	i_beq	0					0									
		1														
	i_bne	0					0									
		1														
	i_lui	x					0									
smem	i_lw	x					0									
	i_sw	x					1									
swb	r_type	x					0									
	i_andi	x					0									
	i_ori	x					0									
	i_lui	x					0									
	i_lw	x					0									

```
module mccu (op, func, z, clock, resetn,
             wpc, wir, wmem, wreg, iord, regrt, m2reg, aluc,
             shift,alusrca, alusrcb, pcsource, jal, sext, state);
```

```
input  [5:0] op, func;
input        z, clock, resetn;
output reg   wpc, wir, wmem, wreg, iord, regrt, m2reg;
output reg [3:0] aluc;
output reg [1:0] alusrcb, pcsource;
output reg       shift, alusrca, jal, sext;
output reg [2:0] state;

reg [2:0]        next_state;
parameter [2:0]  sif  = 3'b000, // IF  state
                 sid  = 3'b001, // ID  state
                 sexe = 3'b010, // EXE state
                 smem = 3'b011, // MEM state
                 swb  = 3'b100; // WB  state

wire r_type,i_add,i_sub,i_and,i_or,i_xor,i_sll,i_srl,i_sra,i_jr;
wire i_addi,i_andi,i_ori,i_xori,i_lw,i_sw,i_beq,i_bne,i_lui,i_j,i_jal;
and(r_type,~op[5],~op[4],~op[3],~op[2],~op[1],~op[0]);
and(i_add,r_type, func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_sub,r_type, func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_and,r_type, func[5],~func[4],~func[3], func[2],~func[1],~func[0]);
and(i_or, r_type, func[5],~func[4],~func[3], func[2],~func[1], func[0]);
and(i_xor,r_type, func[5],~func[4],~func[3], func[2], func[1],~func[0]);
and(i_sll,r_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_srl,r_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_sra,r_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_jr, r_type,~func[5],~func[4], func[3],~func[2],~func[1],~func[0]);
and(i_addi,~op[5],~op[4], op[3],~op[2],~op[1],~op[0]);
and(i_andi,~op[5],~op[4], op[3], op[2],~op[1],~op[0]);
and(i_ori, ~op[5],~op[4], op[3], op[2],~op[1], op[0]);
and(i_xori,~op[5],~op[4], op[3], op[2], op[1],~op[0]);
and(i_lw,   op[5],~op[4],~op[3],~op[2], op[1], op[0]);
and(i_sw,   op[5],~op[4], op[3],~op[2], op[1], op[0]);
and(i_beq, ~op[5],~op[4],~op[3], op[2],~op[1],~op[0]);
and(i_bne, ~op[5],~op[4],~op[3], op[2],~op[1], op[0]);
and(i_lui, ~op[5],~op[4], op[3], op[2], op[1], op[0]);
and(i_j,   ~op[5],~op[4],~op[3],~op[2], op[1],~op[0]);
and(i_jal, ~op[5],~op[4],~op[3],~op[2], op[1], op[0]);
wire i_shift;
or (i_shift,i_sll,i_srl,i_sra);

always @* begin         // control signals' default outputs:
    wpc     = 0;        // do not write pc
    wir     = 0;        // do not write ir
    wmem    = 0;        // do not write memory
    wreg    = 0;        // do not write register file
    iord    = 0;        // select pc as memory address
```

```
aluc      = 4'bx000; // ALU operation: add
alusrca   = 0;       // ALU input a: reg a or sa
alusrcb   = 2'h0;    // ALU input b: reg b
regrt     = 0;       // reg dest no: rd
m2reg     = 0;       // select reg c
shift     = 0;       // select reg a
pcsource  = 2'h0;    // select alu output
jal       = 0;       // not a jal
sext      = 1;       // sign extend

case (state)
    //-------------------------------------------------------- IF:
    sif: begin                           // IF state
        wpc     = 1;                     // write PC
        wir     = 1;                     // write IR
        alusrca = 1;                     // PC
        alusrcb = 2'h1;                  // 4
        next_state = sid;                // next state: ID
    end

    //-------------------------------------------------------- ID:
    sid: begin                           // ID state
        if (i_j) begin                   // j instruction
            pcsource = 2'h3;             // jump address
            wpc  = 1;                    // write PC
            next_state = sif;            // next state: IF
        end else if (i_jal) begin        // jal instruction
            pcsource = 2'h3;             // jump address
            wpc  = 1;                    // write PC
            jal  = 1;                    // reg no = 31
            wreg = 1;                    // save PC+4
            next_state = sif;            // next state: IF
        end else if (i_jr) begin         // jr instruction
            pcsource = 2'h2;             // jump register
            wpc  = 1;                    // write PC
            next_state = sif;            // next state: IF
        end else begin                   // other instructions
            aluc    = 4'bx000;           // add
            alusrca = 1;                 // PC
            alusrcb = 2'h3;              // branch offset
            next_state = sexe;           // next state: EXE
        end
    end

    //-------------------------------------------------------- EXE:
    sexe: begin                          // EXE state
        aluc[3] = i_sra;
```

```
    aluc[2] = i_sub | i_or  | i_srl | i_sra | i_ori  | i_lui;
    aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_beq  |
              i_bne | i_lui;
    aluc[0] = i_and | i_or  | i_sll | i_srl | i_sra  | i_andi |
              i_ori;
    if (i_beq || i_bne) begin   // beq or bne instruction
        pcsource = 2'h1;        // branch address
        wpc  = i_beq & z | i_bne & ~z; // write PC
        next_state = sif;       // next state: IF
    end else begin              // other instruction
        if (i_lw || i_sw) begin // lw or sw instruction
            alusrcb = 2'h2;     // select offset
            next_state = smem;  // next state: MEM
        end else begin          // other instruction
            if (i_shift) shift = 1; // shift instruction
            if (i_addi || i_andi || i_ori || i_xori || i_lui)
                alusrcb = 2'h2; // select immediate
            if (i_andi || i_ori || i_xori) sext=0; // 0-extend
            next_state = swb;   // next state: WB
        end
    end
end

//------------------------------------------------------- MEM:
smem: begin // MEM state
    iord = 1; // memory address = C
    if (i_lw) begin
        next_state = swb;  // next state: WB
    end else begin // store
        wmem = 1;  // write memory
        next_state = sif;  // next state: IF
    end
end

//------------------------------------------------------- WB:
swb: begin // WB state
    if (i_lw) m2reg = 1; // select memory data
    if (i_lw || i_addi || i_andi || i_ori || i_xori || i_lui)
        regrt = 1;       // reg dest no: rt
    wreg = 1;            // write register file
    next_state = sif;    // next state: IF
end
//------------------------------------------------------- END

default: begin
    next_state = sif;    // default state
end
```

```
        endcase
    end

    always @ (posedge clock or negedge resetn) begin // state registers
        if (resetn == 0) begin
            state <= sif;
        end else begin
            state <= next_state;
        end
    end
endmodule
```

7.4 存储器及测试程序设计

7.4.1 存储器设计

```
module mcmem (clk, dataout, datain, addr, we, inclk, outclk);
    input [31:0] datain;
    input [31:0] addr;
    input        clk, we, inclk, outclk;
    output [31:0] dataout;

    wire          write_enable = we & ~clk;
    lpm_ram_dq ram (.data(datain),.address(addr[7:2]),
                    .we(write_enable),.inclock(inclk),
                    .outclock(outclk),.q(dataout));
    defparam   ram.lpm_width    = 32;
    defparam   ram.lpm_widthad = 6;
    defparam   ram.lpm_indata   = "registered";
    defparam   ram.lpm_outdata = "registered";
    defparam   ram.lpm_file     = "mcmem.mif";
    defparam   ram.lpm_address_control = "registered";
endmodule
```

7.4.2 测试程序代码

因为只有一个存储器模块，所以测试程序和数据必须放在一起。

```
DEPTH = 64;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
[0..3F] : 00000000; % Range--Every address from 0 to 3F = 00000000        %
```

```
 0 : 3c010000; % (00) main:  lui  r1, 0           # address of data[0]     %
 1 : 34240080; % (04)        ori  r4, r1, 0x80    # address of data[0]     %
 2 : 20050004; % (08)        addi r5, r0,  4      # counter                %
 3 : 0c000018; % (0c) call:  jal  sum             # call function          %
 4 : ac820000; % (10)        sw   r2, 0(r4)       # store result           %
 5 : 8c890000; % (14)        lw   r9, 0(r4)       # check sw               %
 6 : 01244022; % (18)        sub  r8, r9, r4      # sub: r8 <-- r9 - r4    %
 7 : 20050003; % (1c)        addi r5, r0,  3      # counter                %
 8 : 20a5ffff; % (20) loop2: addi r5, r5, -1      # counter - 1            %
 9 : 34a8ffff; % (24)        ori  r8, r5, 0xffff # zero-extend: 0000ffff  %
 A : 39085555; % (28)        xori r8, r8, 0x5555 # zero-extend: 0000aaaa  %
 B : 2009ffff; % (2c)        addi r9, r0, -1      # sign-extend: ffffffff  %
 C : 312affff; % (30)        andi r10, r9,0xffff # zero-extend: 0000ffff  %
 D : 01493025; % (34)        or   r6, r10, r9     # or:  ffffffff          %
 E : 01494026; % (38)        xor  r8, r10, r9     # xor: ffff0000          %
 F : 01463824; % (3c)        and  r7, r10, r6     # and: 0000ffff          %
10 : 10a00001; % (40)        beq  r5, r0, shift  # if r5 = 0, goto shift  %
11 : 08000008; % (44)        j    loop2           # jump loop2             %
12 : 2005ffff; % (48) shift: addi r5, r0, -1      # r5   = ffffffff        %
13 : 000543c0; % (4c)        sll  r8, r5, 15      # <<15 = ffff8000        %
14 : 00084400; % (50)        sll  r8, r8, 16      # <<16 = 80000000        %
15 : 00084403; % (54)        sra  r8, r8, 16      # >>16 = ffff8000(arith) %
16 : 000843c2; % (58)        srl  r8, r8, 15      # >>15 = 0001ffff(logic) %
17 : 08000017; % (5c) finish: j   finish          # dead loop              %
18 : 00004020; % (60) sum:   add  r8, r0, r0      # sum                    %
19 : 8c890000; % (64) loop:  lw   r9, 0(r4)       # load data              %
1A : 20840004; % (68)        addi r4, r4,  4      # address + 4            %
1B : 01094020; % (6c)        add  r8, r8, r9      # sum                    %
1C : 20a5ffff; % (70)        addi r5, r5, -1      # counter - 1            %
1D : 14a0fffb; % (74)        bne  r5, r0, loop   # finish?                %
1E : 00081000; % (78)        sll  r2, r8, 0       # move result to v0      %
1F : 03e00008; % (7c)        jr   r31             # return                 %
20 : 000000A3; % (80) data[0] %
21 : 00000027; % (84) data[1] %
22 : 00000079; % (88) data[2] %
23 : 00000115; % (8c) data[3] %
24 : 00000000; % (90) sum     %
END ;
```

7.4.3 多周期 CPU 测试结果

图 7.13 ~ 图 7.15 给出了部分仿真波形。为了方便检查，我们把每条指令所处的状态号 q 也显示出来了。注意，q = 0 表示 IF 状态，正在根据 PC 取指令。IF 状态结束时，由时钟上升沿把指令存入指令寄存器 (IR)。IR 的内容一直保持到打入下一条指令为止。

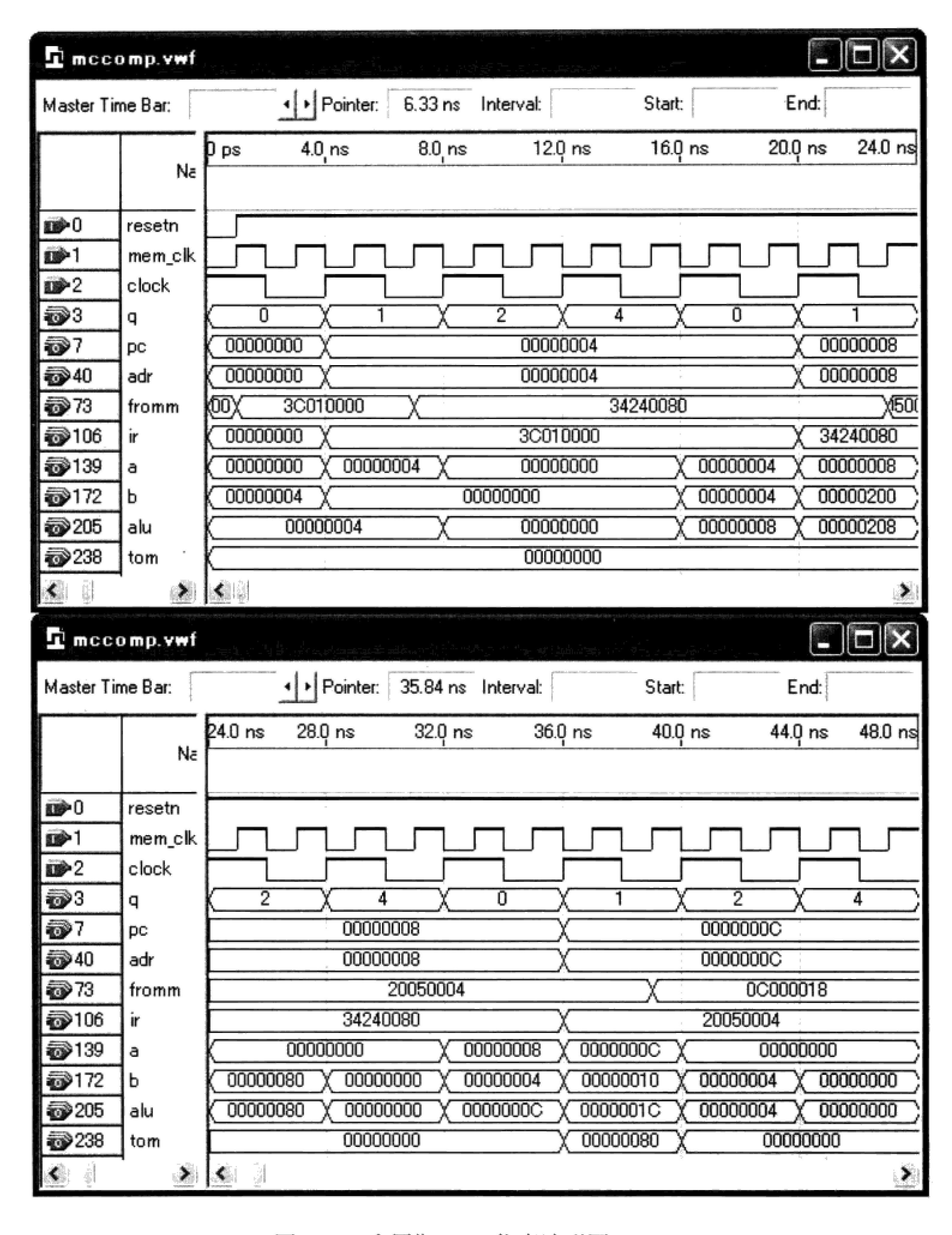


图 7.13　多周期 CPU 仿真波形图 (1 ～ 2)

图 7.13 中的第一条指令是 ALU 计算类型的指令 (lui)，它需要 4 个周期，状态号为 0、1、2 和 4，分别代表 IF、ID、EXE 和 WB。在 IF 状态，PC = 0，从存储器取来指令，由 4ns 处的 Clock 上升沿把指令打入 IR，同时把 PC + 4 打入 PC。

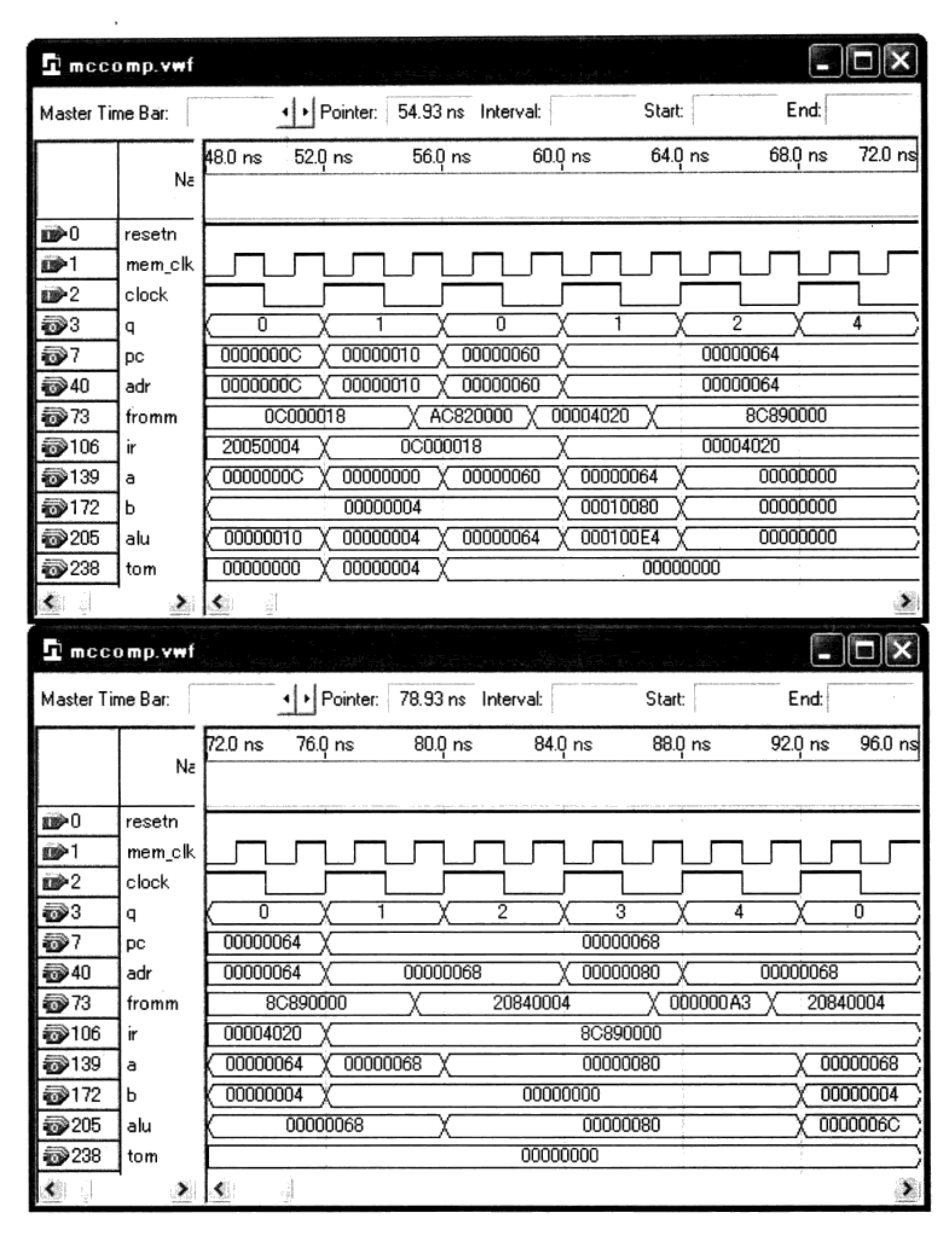


图 7.14 多周期 CPU 仿真波形图 (3 ～ 4)

图 7.14 上半部分 PC = 0000000C 处的指令 (0C000018) 是子程序调用指令 jal，它只需两个周期，转到 PC = 00000060。图 7.14 下半部分中 PC = 00000064 处的指令是 lw，它需要最长的 5 个周期。

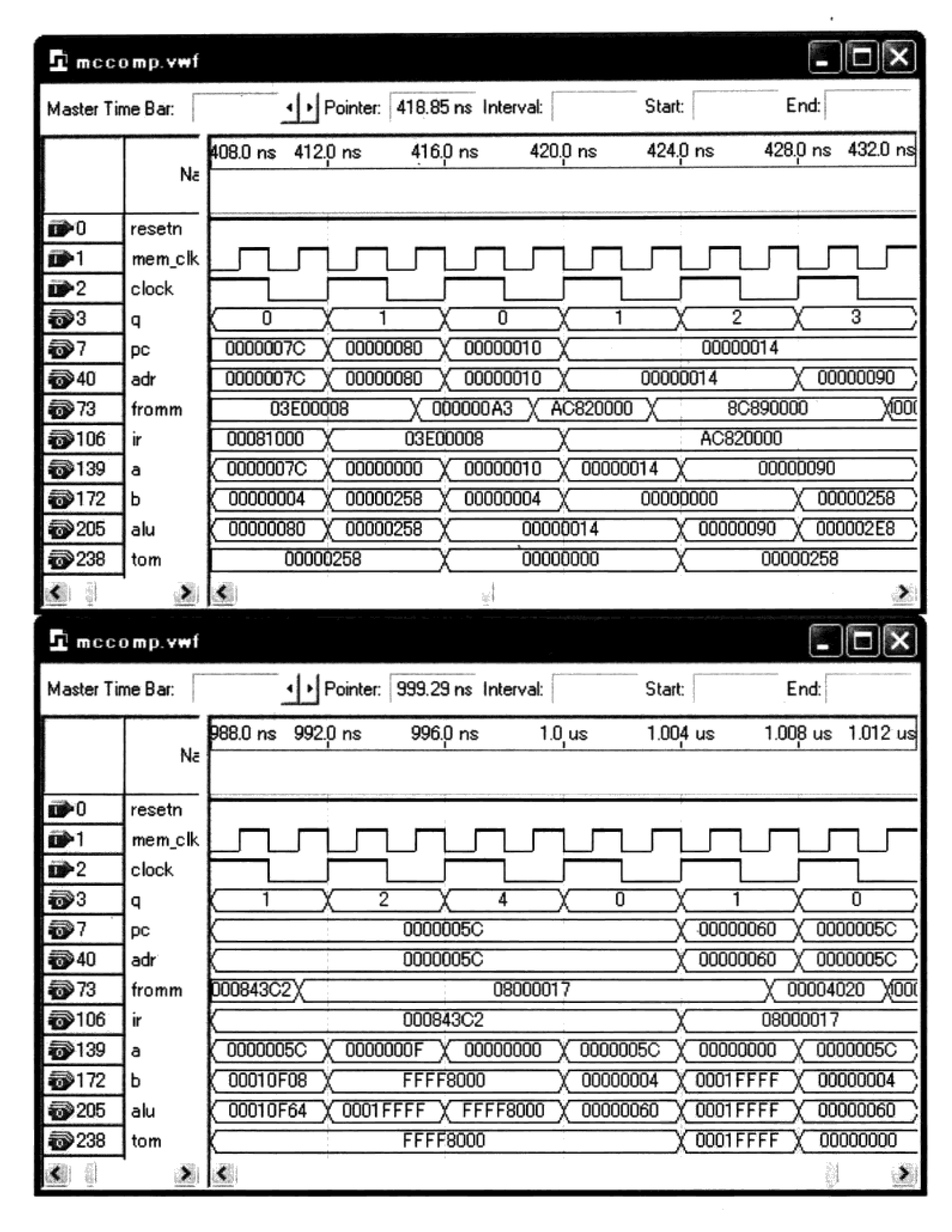
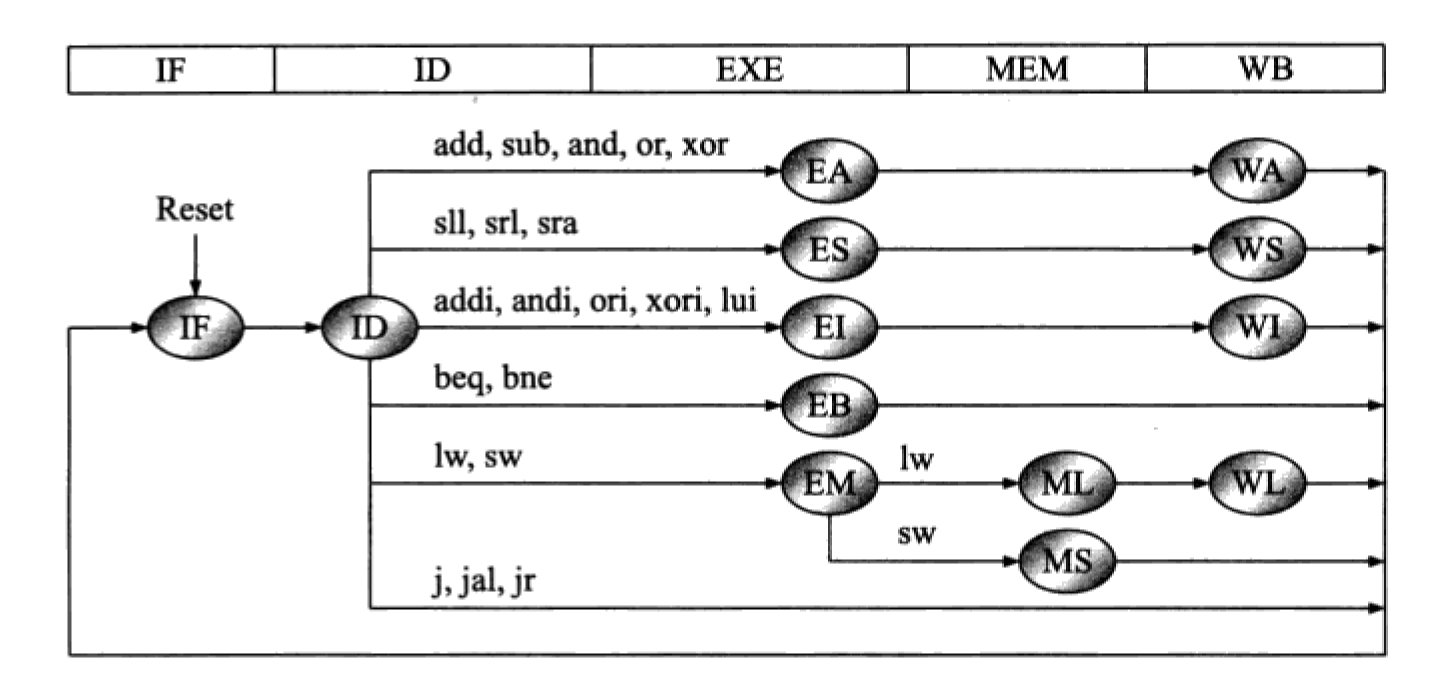


图 7.15　多周期 CPU 仿真波形图 (5 ～ 6)

图 7.15 上半部分 PC = 0000007C 处的指令 (03E00008) 是返回指令 jr，它只需两个周期 (IF 和 ID)。在 ID 周期结束时，返回到 PC = 00000010 处。下半部分是测试程序最后一条 j 指令跳转到自己处 (PC = 0000005C) 的波形。

7.5 习题

1. 试比较单周期 CPU 与多周期 CPU 各自的优缺点。
2. 图 7.10 中的寄存器 A、B 和 DR 是否可以不用？
3. 用逻辑图输入的方法设计图 7.10 的 CPU。
4. 试用以下状态转移图设计多周期 CPU。



5. 设计一个多周期 CPU，使 lui 指令用两个周期完成。
6. 试书写 Verilog HDL 代码，以完全彻底的功能描述风格实现多周期 CPU 的设计并给出仿真波形。
7. 我们在叙述单周期 CPU 与多周期 CPU 的设计方法时使用了相同的测试程序并给出了测试结果，从而我们知道了两种 CPU 在执行相同的测试程序时所用的时间。假设多周期 CPU 的一个时钟周期是 4ns，而单周期 CPU 的一个时钟周期比多周期 CPU 的一个时钟周期的 5 倍略短一些，比如 19ns。试从执行时间上比较两种 CPU 执行测试程序时的性能。
8. 大作业：设计一个能处理异常和中断的多周期 CPU 使其能够处理诸如结果溢出等异常及外部的中断请求，并写出一份像样的报告[1]。友情提示：在状态转移图中加入一些状态、专门应付异常和中断。

[1]致仍在使用 MS Word 的读者：建议大家以后写论文时尽量少用 MS Word，多用 LaTeX。用 Word 既花钱效果又差；用 LaTeX 既免费效果又好，何乐而不为呢？作者见到用 Word 写的论文，不管是本科毕业设计论文、硕士论文、博士论文还是投到国际会议或杂志的论文，排版很差，尤其是公式，图的质量也很差，第一感觉就很不舒服，还没读具体的内容，头就先大了。有这种感觉的绝非作者一个人，你想，连微软公司的人写文章都用 LaTeX。从我做起，从现在做起，从课题报告做起。即使你没有 Linux 环境，在 Windows 下也有各种方法可以安装：上网去找。如果你已经用了，稍微表扬自己一下。

第 8 章　流水线 CPU 及其 Verilog HDL 设计

本章介绍流水线 CPU 的设计以及精确中断的实现方法并给出 Verilog HDL 代码。本章介绍的流水线 CPU 具有以下特点：(1) 使用内部前推 (Internal Forwarding) 技术解决数据相关问题；(2) 使用延迟转移 (Delayed Branch) 技术解决控制相关问题；(3) 暂停流水线解决存储器访问的数据相关问题。

8.1　流水线技术的基本概念

几乎没有例外，现代化的工厂都采用流水线技术生产产品。试设想有一个汽车制造厂，不采用流水线技术，等一辆汽车生产完毕，再从头开始制造下一辆汽车，这样的工厂一年能生产几辆车？

如果把 CPU 执行一条指令看成工厂生产一辆汽车，我们也可以使用流水线技术：不要等一条指令执行完成后再执行下一条指令，而是把一条指令的执行分成若干“级” (Stage)，不同的指令的不同的级可以在同一个周期同时执行。

为了叙述方便，我们假设有一个 CPU 只能执行算术运算和存储器访问指令。以下的汇编程序把一个数组 x 的 4 个数据都加上一个标量 s (假设 s 在寄存器 r10 中)。汇编程序共有 12 条指令，左边是地址，右边是注释，中间是指令。

```
# addr  instructions     comments
  100:  lw  r2, 00(r1)   # r2 <-- mem[r1+00]; load x[0]
  104:  lw  r3, 04(r1)   # r3 <-- mem[r1+04]; load x[1]
  108:  lw  r4, 08(r1)   # r4 <-- mem[r1+08]; load x[2]
  112:  lw  r5, 12(r1)   # r5 <-- mem[r1+12]; load x[3]
  116:  add r6, r2, r10  # r6 <-- r2 + r10; x[0] = x[0] + s
  120:  add r7, r3, r10  # r7 <-- r3 + r10; x[1] = x[1] + s
  124:  add r8, r4, r10  # r8 <-- r4 + r10; x[2] = x[2] + s
  128:  add r9, r5, r10  # r9 <-- r5 + r10; x[3] = x[3] + s
  132:  sw  r6, 00(r1)   # mem[r1+00] <-- r6; store x[0]
  136:  sw  r7, 04(r1)   # mem[r1+04] <-- r7; store x[1]
  140:  sw  r8, 08(r1)   # mem[r1+08] <-- r8; store x[2]
  144:  sw  r9, 12(r1)   # mem[r1+12] <-- r9; store x[3]
```

图 8.1 是单周期 CPU 的电路。图中所示的是 CPU 执行第 1 条 lw r2, 00(r1) 指令的情形。执行该指令的步骤如下。

1) IF：程序计数器 PC 的值为 100；从指令存储器取出的是 lw 指令；
2) ID：从寄存器 r1 读出数据；立即数符号扩展；
3) EXE：计算存储器地址：ALU 做加法 (ALU 的 b 输入端选择扩展后的立即数)；
4) MEM：使用计算好的地址访问存储器，从中读出一个 32 位的数据；
5) WB：最后把该数据写入寄存器堆中的 r2 寄存器 (时钟上升沿写入)。

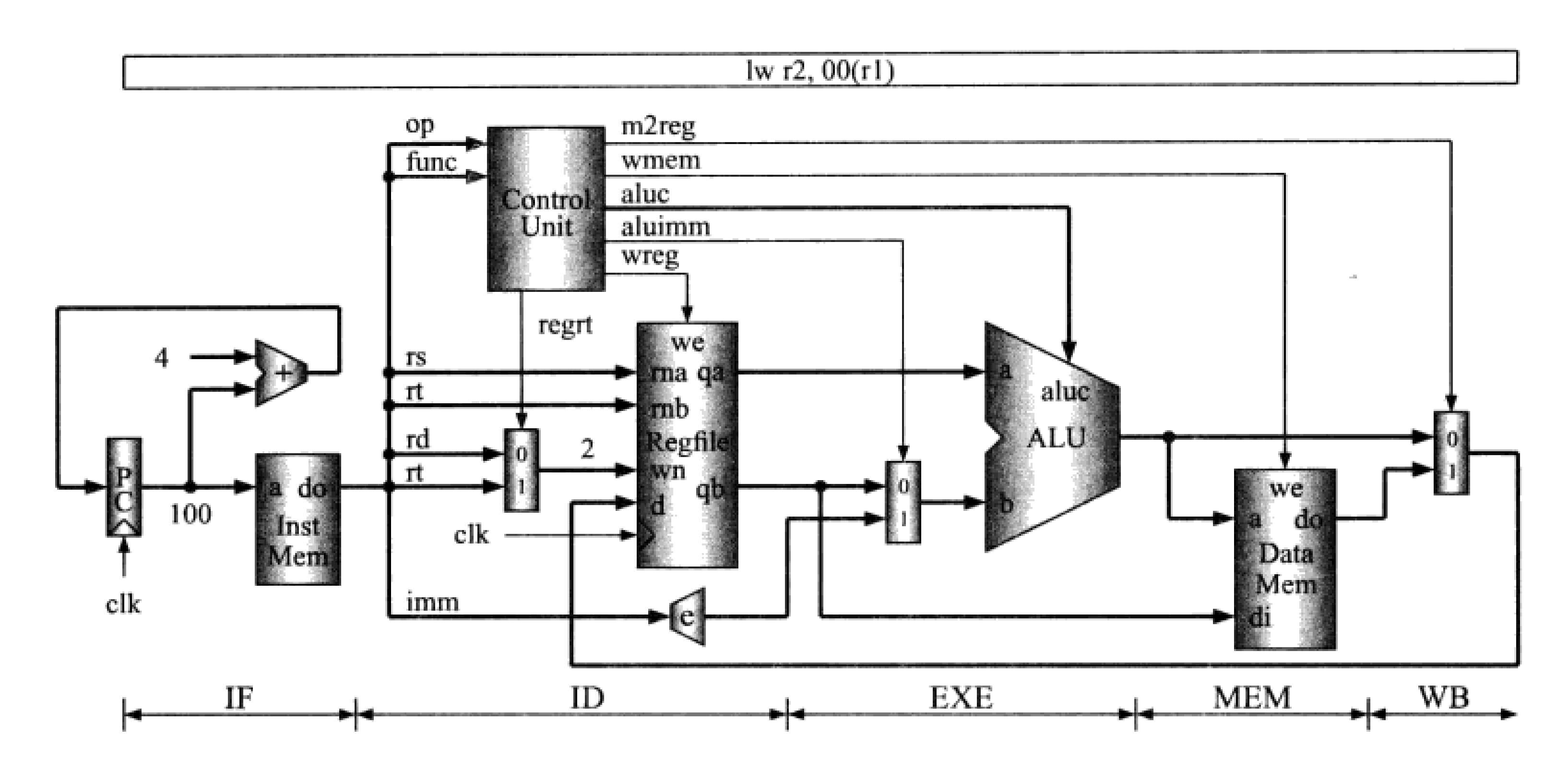


图 8.1 单周期 CPU 执行 lw 指令

假设每个步骤都用 1ns，则单周期 CPU 的周期长度为 5ns，执行 12 条指令共需 $5 \times 12 = 60$ns。如果是多周期 CPU，周期长度为 1ns，lw 指令用 5 个周期，add 和 sw 指令各用 4 个周期，则执行以上 12 条指令共需 $5 \times 4 + 4 \times 8 = 52$ns。图 8.2 示出流水线 CPU 的时序以及单周期和多周期 CPU 的时序。

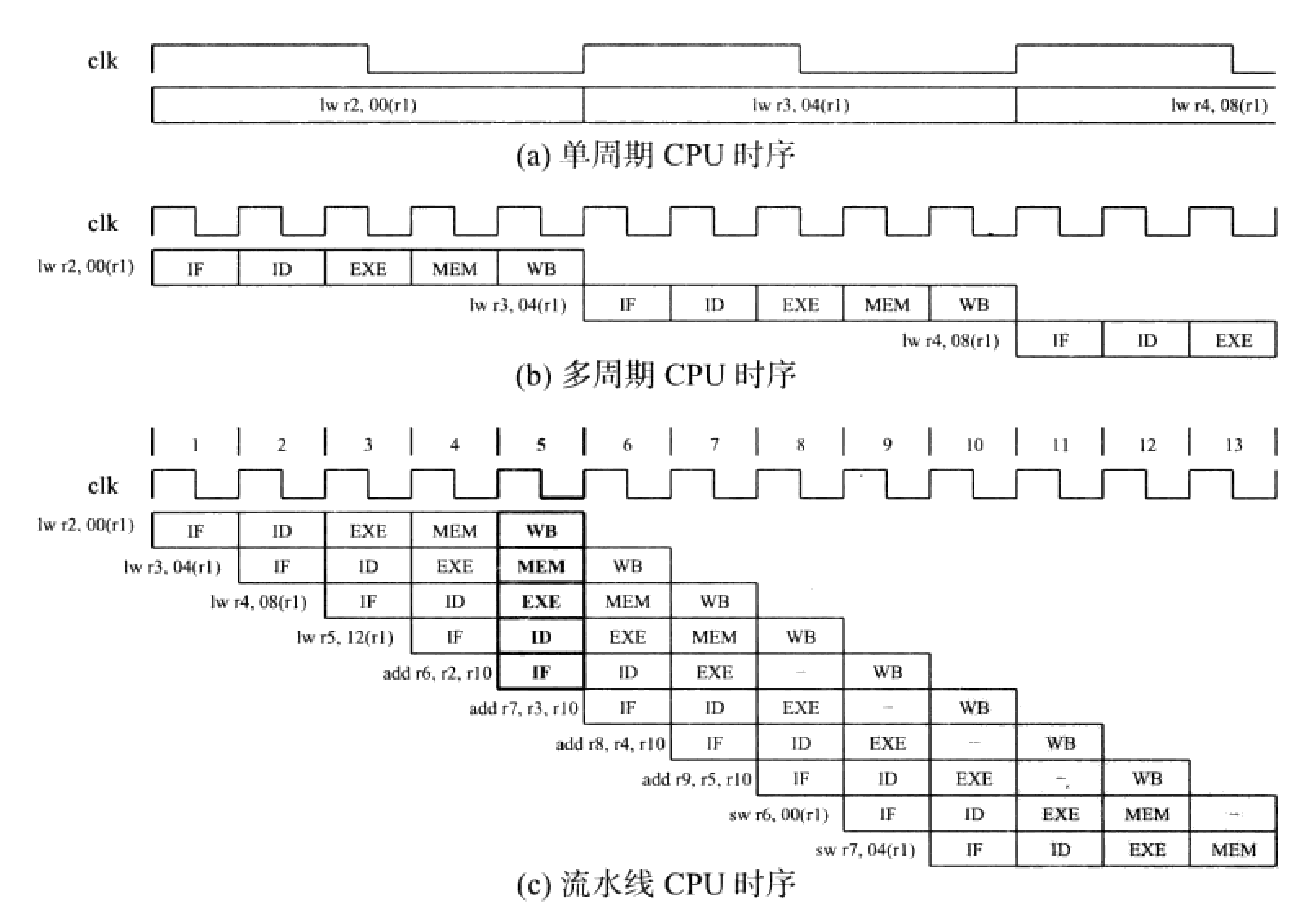


图 8.2 流水线 CPU 与单周期/多周期 CPU 的时序比较

流水线 CPU 执行以上 12 条指令共需 12 + 5 − 1 = 16ns。当指令充满流水线时，如图 8.2 中的第 5 个周期 (粗线及粗黑体部分)，5 条指令同时在执行，不同的指令处在不同的级 (重叠执行)。

由于在一个周期有多条指令同时执行，我们必须把每一级的执行结果暂时保存起来，以便在下一级使用。因此，在流水线 CPU 中，除了 PC 寄存器，我们还要在每级结束的地方使用寄存器。这些寄存器称为流水线寄存器 (Pipeline Registers)。

我们的流水线 CPU 共有 5 级，它们分别是：(1) 取指令 IF 级；(2) 指令译码 ID 级；(3) 指令执行 EXE 级；(4) 存储器访问 MEM 级；(5) 结果写回 WB 级。以下对这 5 级所需的电路分别加以描述。

8.1.1 取指令 IF 级的电路

取指令 IF 级的电路如图 8.3 所示。它也是我们的流水线 CPU 的第 1 个周期：正在取第 1 条指令。这就像一所新建的大学，只有一年级的“新生”。

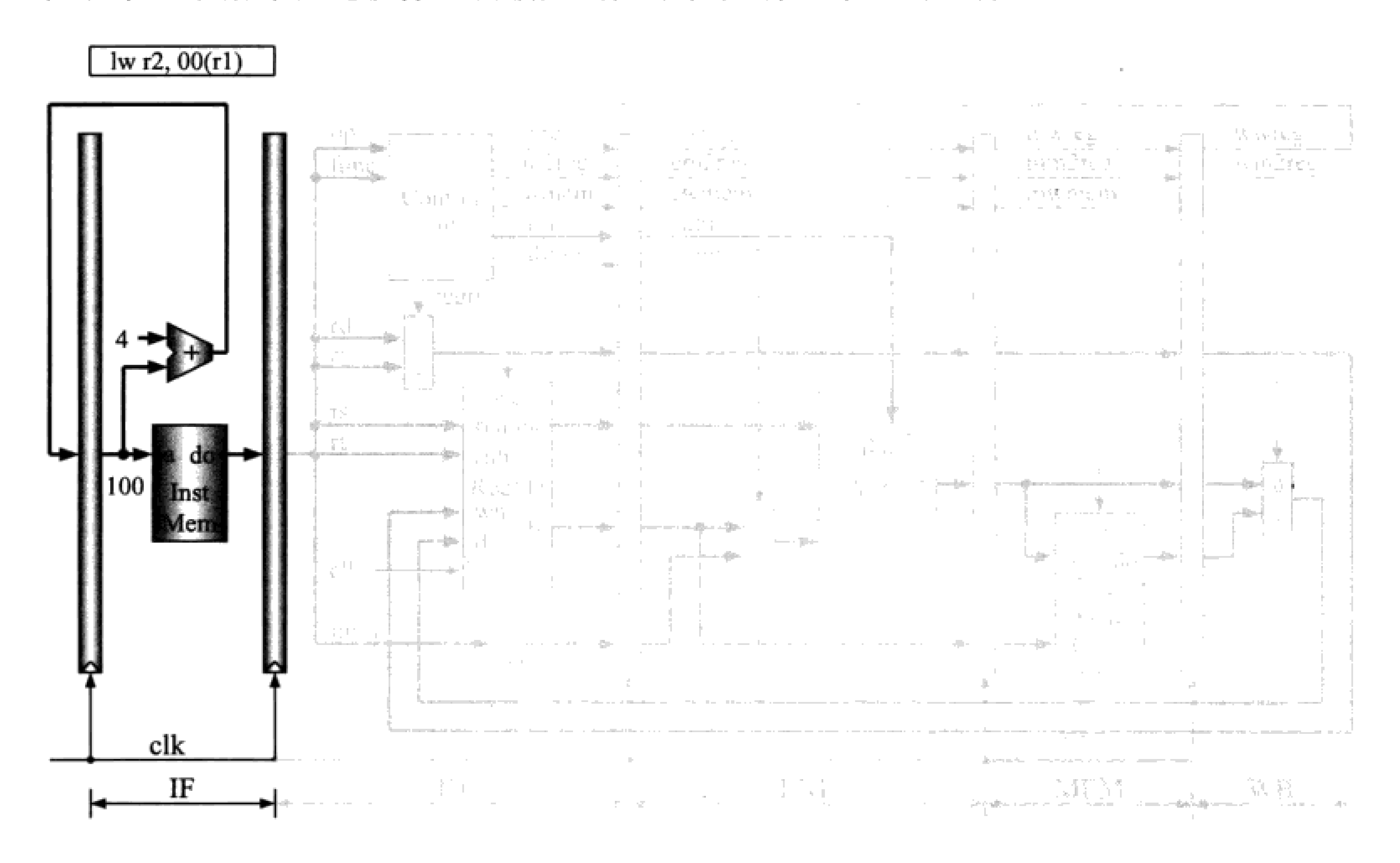


图 8.3 流水线 CPU 的 IF 级 (第 1 个周期)

最左边的流水线寄存器是程序计数器 PC，第 2 个是指令寄存器 IR。在第 1 个周期结束时 (时钟上升沿处)，把从指令存储器取出的指令写入 IR (从一年级升入二年级)，与此同时，把 PC + 4 写入 PC (招收新生)。

注意，实际的 CPU 可以执行转移和跳转指令，并非只是 PC + 4。我们将在后面描述流水线 CPU 如何执行转移和跳转指令。

8.1.2 指令译码 ID 级的电路

第 1 条指令进入 ID 级 (第 2 个周期)，如图 8.4 所示。在第 2 个周期有两项工作同时在做：对第 1 条指令译码和从指令存储器取第 2 条指令。两条指令在图的上方标出，即第 1 条指令处在 ID 级、第 2 条指令处在 IF 级。

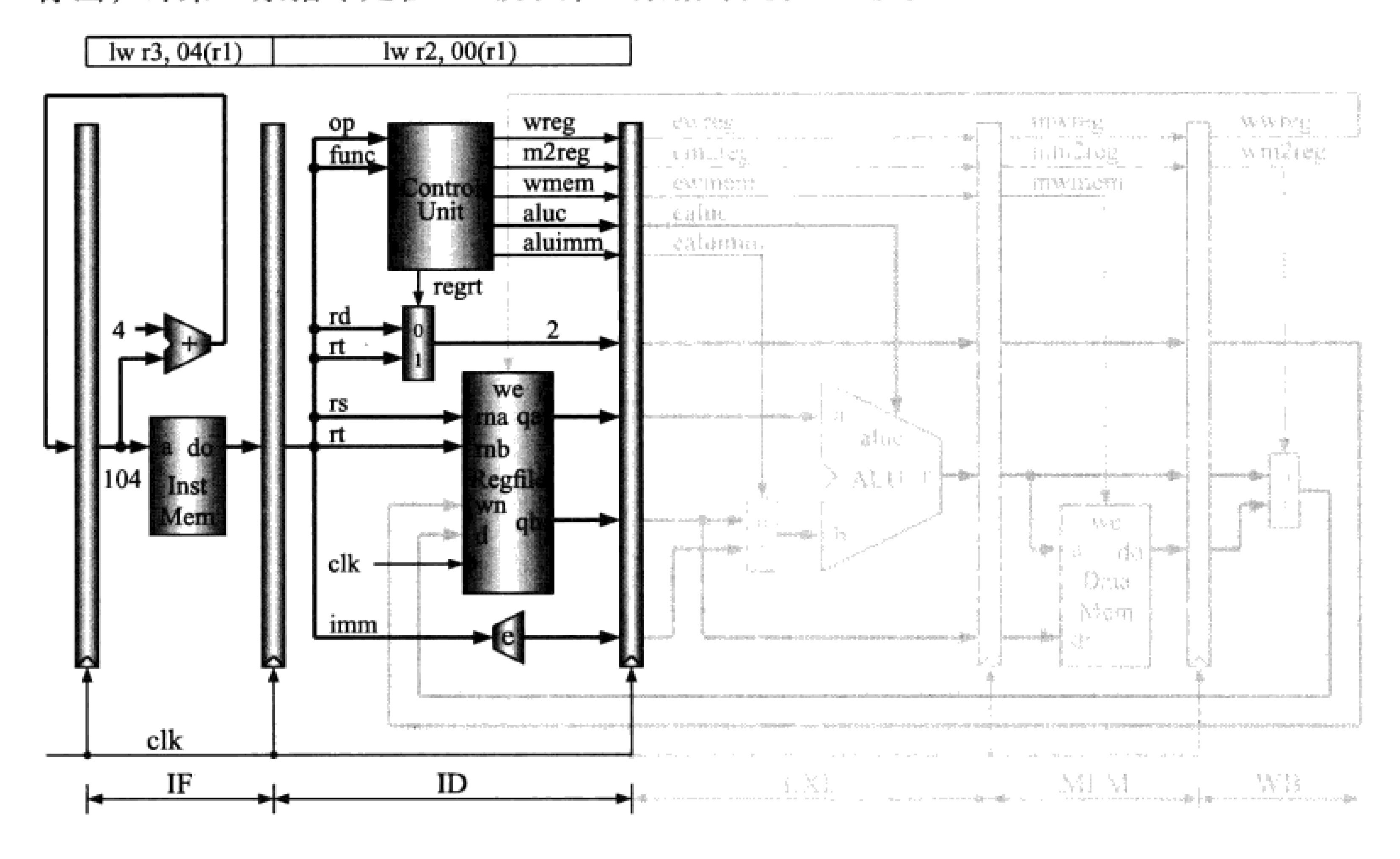


图 8.4 流水线 CPU 的 ID 级 (第 2 个周期)

处在 ID 级的第 1 条指令由 IR 送出。根据指令中的 rs 和 rt，两个 32 位的数据从寄存器堆读出。与此同时，指令中的 16 位的立即数也被扩展为 32 位。注意，lw 指令并不使用从 rt 寄存器读出的数据。对该指令的译码由控制部件完成。控制信号的意义如下。

1) regrt 为 1 时选择 rt 作为目的寄存器号，为 0 时选择 rd；
2) aluimm 为 1 时选择立即数送给 ALU 的 b 输入端，为 0 时选择寄存器数据；
3) aluc 是 ALU 的操作控制码；
4) wmem 为 1 时写存储器，为 0 时不写；
5) m2reg 为 1 时选择存储器数据，为 0 时选择 ALU 的计算结果；
6) wreg 为 1 时把结果写入寄存器堆，为 0 时不写。

除了 regrt 只在本级使用外，其他控制信号要被写入流水线寄存器，以备后续周期使用。因为当前指令为 lw，所以 regrt = 1，aluimm = 1，aluc = X000，wmem = 0，m2reg = 1，wreg = 1。在第 2 个周期结束时，3 个流水线寄存器同时保存数据 (当然也包括控制信号)。

8.1.3 指令执行 EXE 级的电路

第 1 条指令进入 EXE 级 (第 3 个周期)，如图 8.5 所示。在 EXE 级，ALU 为 lw 指令计算存储器地址；该级使用两个控制信号：ealuimm 选择立即数、ealuc 告诉 ALU 做加法。在 EXE 级的控制信号的名称前面都加了一个字母 e，表示是 EXE 级的信号，以便与 ID 级的相应信号区别开来。

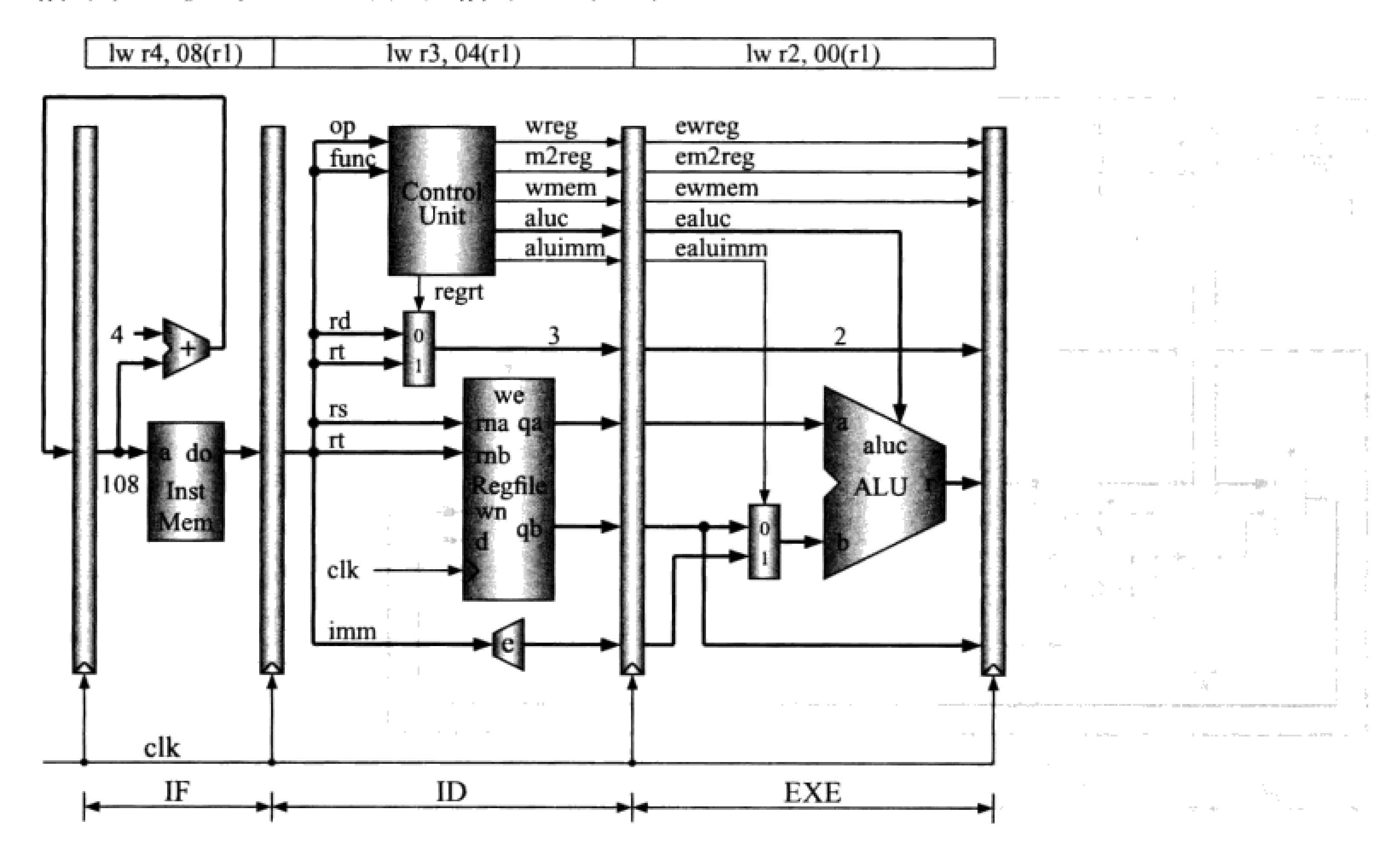


图 8.5 流水线 CPU 的 EXE 级 (第 3 个周期)

第 2 条指令处在 ID 级，正在译码；IF 级正在取第 3 条指令。同样，在第 3 个周期结束时，4 个流水线寄存器同时保存数据。

8.1.4 存储器访问 MEM 级的电路

第 1 条指令进入 MEM 级 (第 4 个周期)，如图 8.6 所示。其他指令也都前进了一级。由于是 lw 指令，该级读数据存储器，控制信号 mwmem 为 0 (不写存储器)。在 MEM 级的控制信号的名称前面都加了一个字母 m。在第 4 个周期结束时，5 个流水线寄存器同时保存数据。

8.1.5 结果写回 WB 级的电路

第 1 条指令进入 WB 级 (第 5 个周期)，如图 8.7 所示。控制信号的名称前面都加了一个 w。这时的控制信号 wm2reg 为 1，wwreg 也为 1，目的寄存器号为 2，已经准备好了保存从数据存储器取出的数据，就等着时钟上升沿的到来了。

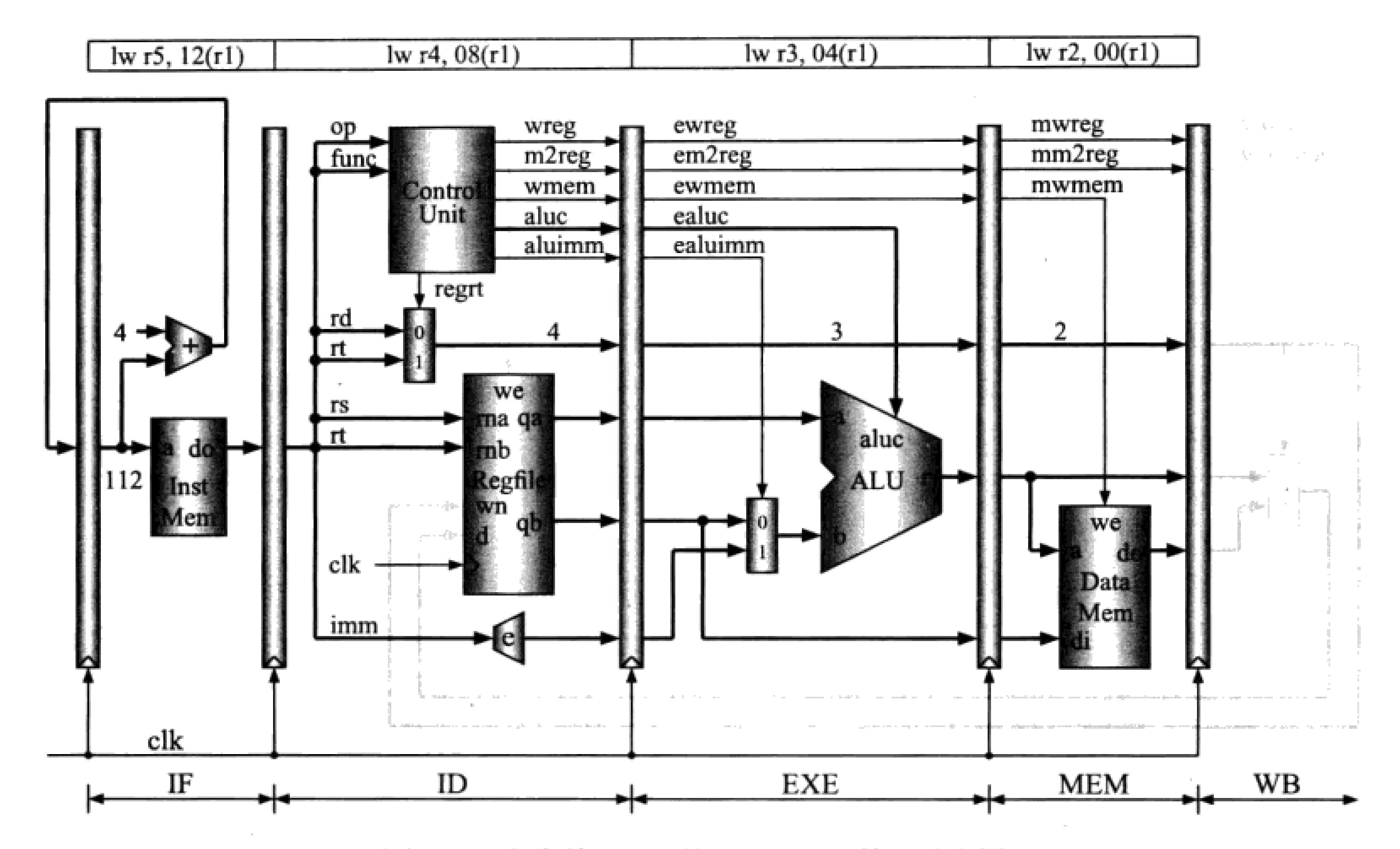


图 8.6　流水线 CPU 的 MEM 级 (第 4 个周期)

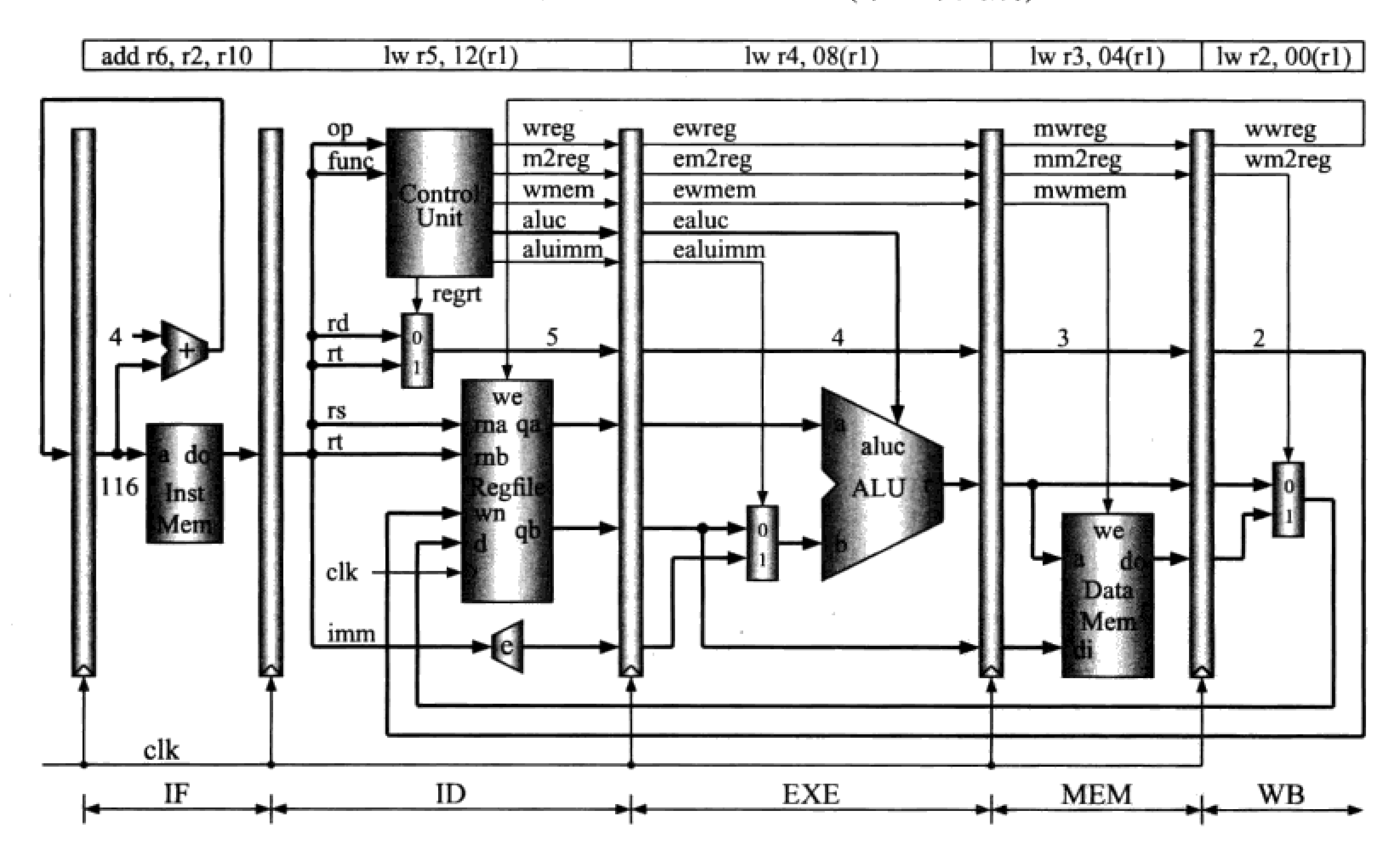


图 8.7　流水线 CPU 的 WB 级 (第 5 个周期)

这时的 5 级流水线均充满了指令。时钟上升沿到来时，把数据写入寄存器 r2。这样，第 1 条 lw 指令就全部执行完毕。以后就是每个周期有一条指令“毕业”，同时有一条指令“报到”，“在校”的指令有 5 个“年级”。

8.2 流水线 CPU 的相关问题及解决对策

是不是流水线 CPU 的设计就像第一节讨论的那样简单呢？当然不是。流水线 CPU 的设计有三大著名的问题需要解决。这三大问题是 (1) 结构相关 (Structural Hazard / Dependence)、(2) 数据相关 (Data Hazard / Dependence) 和 (3) 控制相关 (Control Hazard / Dependence)。控制相关也称转移相关 (Branching Hazard / Dependence)。

结构相关是指流水线 CPU 在同时执行多条指令时争用硬件资源而引起的冲突。一个典型的例子是，假设流水线 CPU 只使用一个存储器模块来存放指令和数据，而且该存储器模块不支持两个同时的访问。lw 或 sw 指令在 MEM 级访问存储器，而流水线 CPU 在每个周期都要取出一条指令。也就是说在一个周期有两个存储器访问，这就造成了资源冲突。

解决结构相关的方法比较简单。因为现在的 VLSI 技术能够在一个芯片内集成大量的晶体管，所以我们可以本着“缺什么补什么”的原则避免出现资源冲突。例如我们可以在流水线 CPU 芯片的内部集成分开的指令 Cache 和数据 Cache，使得取指令和访问数据存储器能够同时进行。本节重点讨论数据相关和控制相关两大问题。

8.2.1 数据相关及解决对策

图 8.8 是流水线 CPU 的另一种画法，强调 WB 级结束时结果写回寄存器堆。

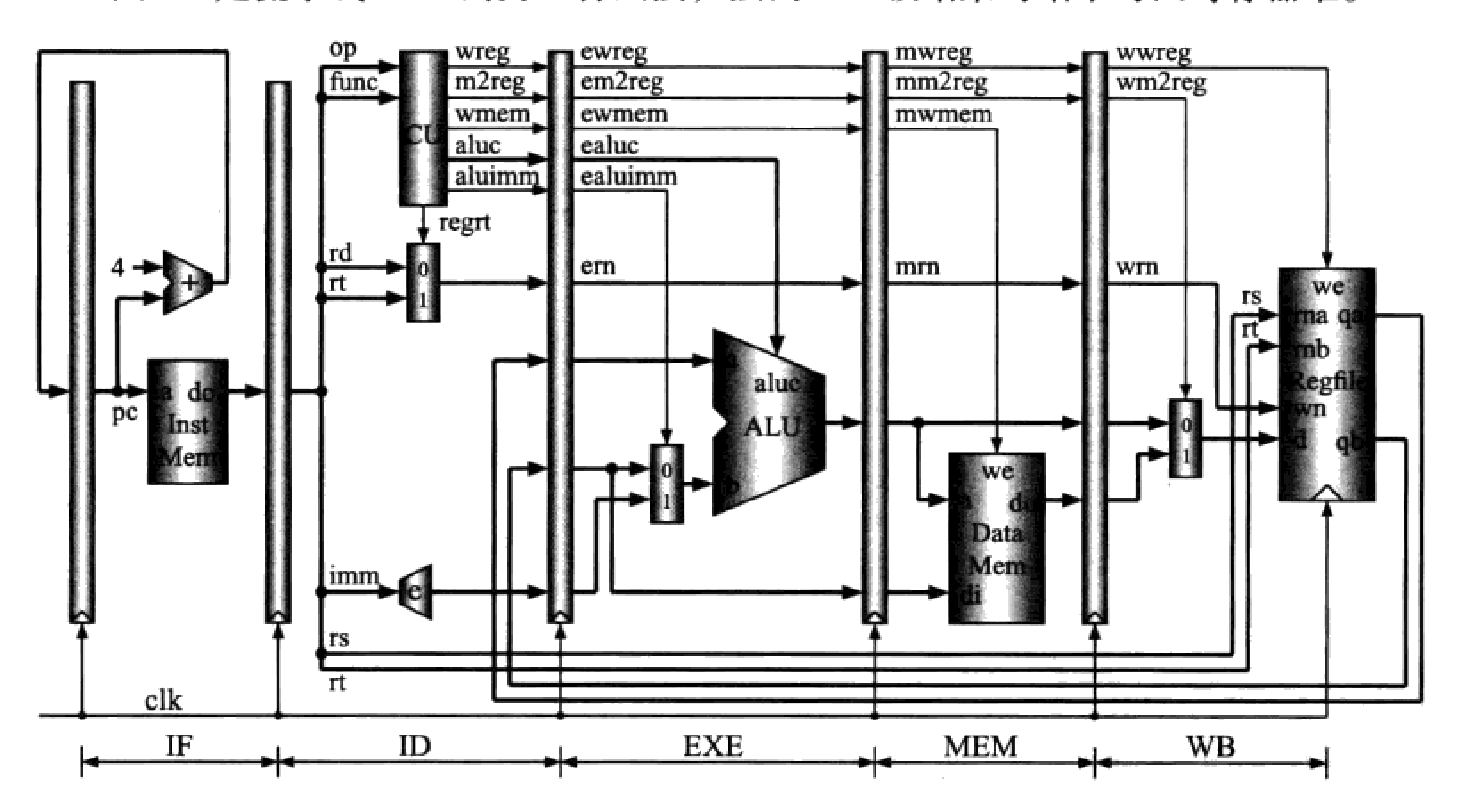


图 8.8 流水线 CPU 的另一种画法

由于流水线 CPU 同时执行多条指令，指令之间会出现数据相关的问题，即一条指令还没有执行完，而它的后续指令要使用它的结果。我们看图 8.9 的例子。

第 1 条指令 add r3, r1, r2 在第 5 个周期结束时把相加结果写入寄存器 r3。后续的指令 sub、or 和 xor 在各自的 ID 级不能从寄存器 r3 读出正确的结果。

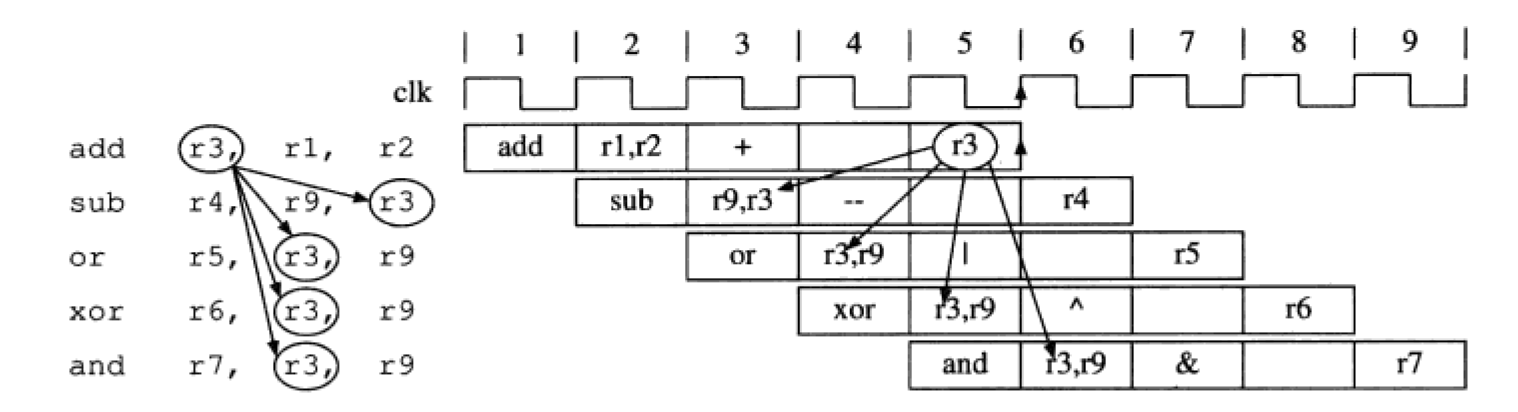


图 8.9 数据相关的例子

如何解决这个数据相关问题？停流水线等待？不停！我们知道，第 1 条指令 add 是由 ALU 实现加法操作。那么第 2 条指令 sub 由谁实现减法操作？也是 ALU。这就意味着 ALU 做减法时，加法已经完成。我们可以把刚刚出炉的新鲜的加法结果直接送给下一条指令去吃。这就是所谓内部前推 (Internal Forwarding)，或称内部旁路 (Bypass)。我们的做法是把内部前推由 EXE 级移到 ID 级，如图 8.10 和图 8.11 所示。关键部件是 ID 级的两个四选一多路器，请检查它们的输入都来自何方。

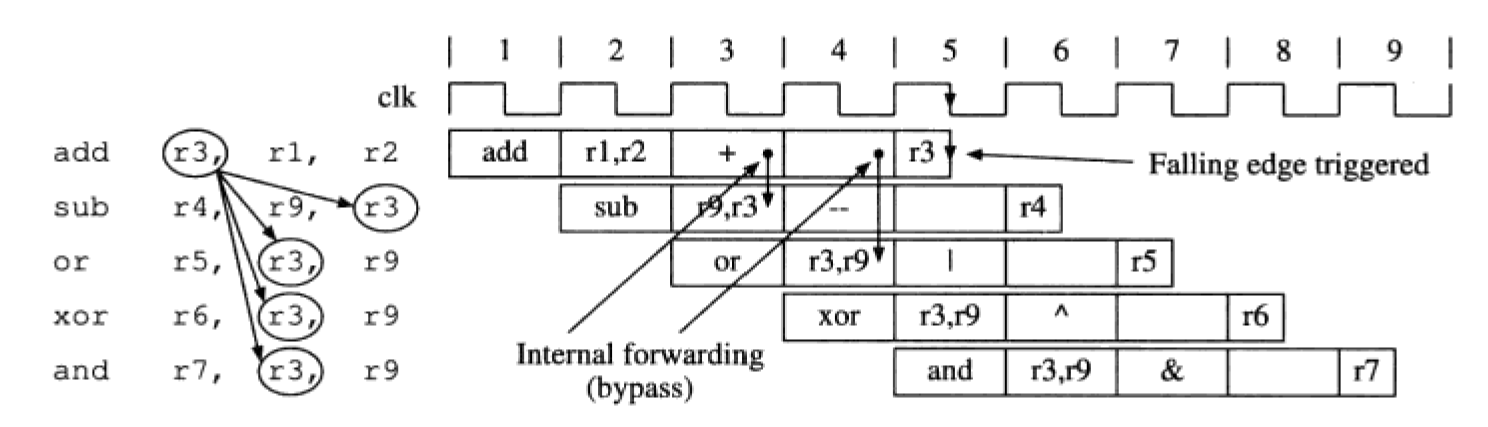


图 8.10 解决数据相关问题 —— 内部前推

内部前推解决后续两条指令的数据相关问题。我们用时钟的下降沿在 WB 级保存结果 (使用图 8.11 中的非门)，即 WB 级只用半个周期。内部前推由两个多路选择器实现，它们的控制信号 fwda 和 fwdb 用来选择合适的数据，在 ID 级结束时写入流水线寄存器。以下是产生 fwda 和 fwdb 的 Verilog HDL 代码。注意代码的次序：先判断离它最近的指令。为什么呢？

```
always @ (ewreg, mwreg, ern, mrn, em2reg, mm2reg, rs, rt) begin
    fwda = 2'b00; // default forward a: no hazards
    if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
        fwda = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
            fwda = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                fwda = 2'b11; // select mem_lw
            end
```

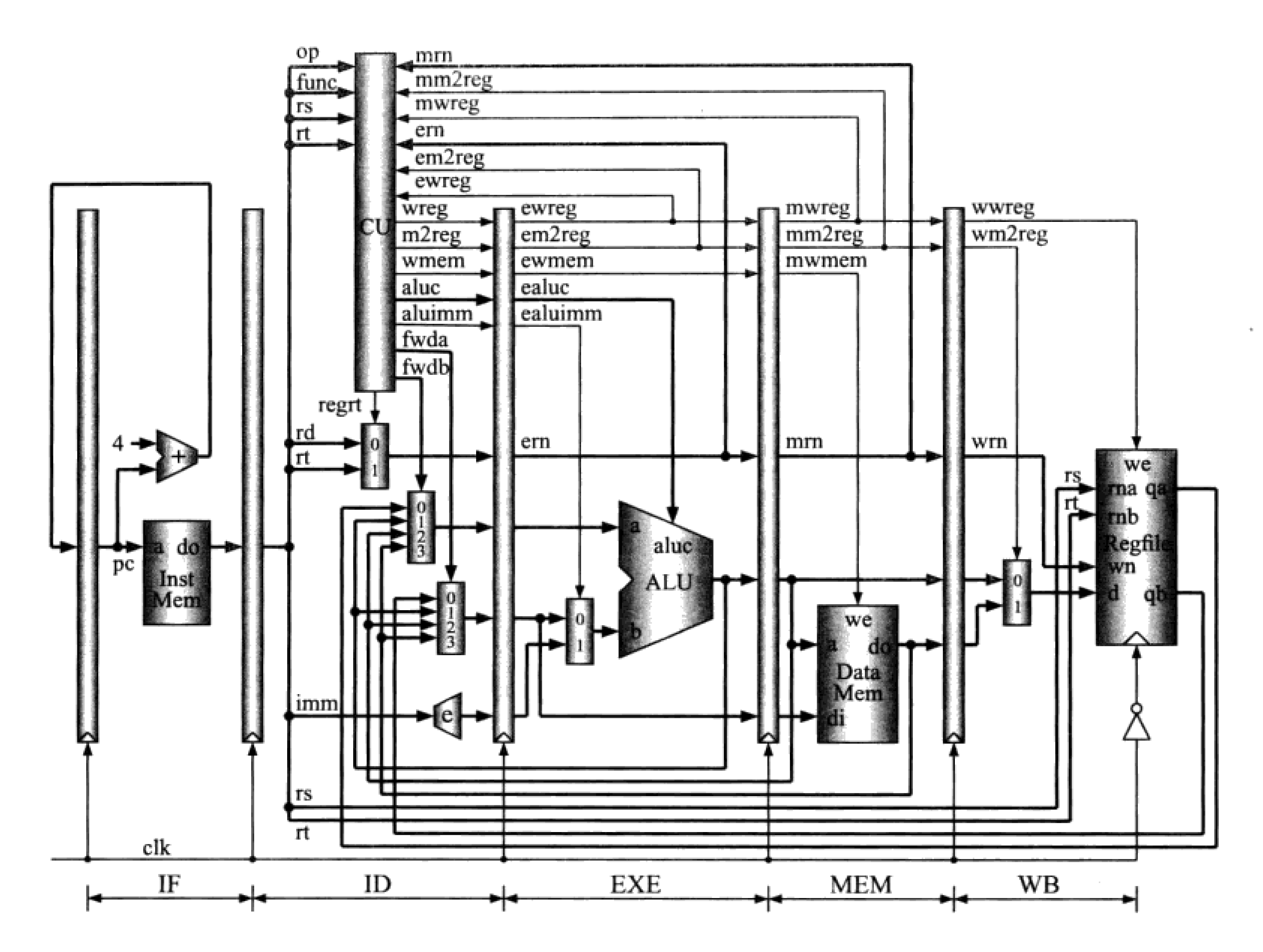


图 8.11 带有内部前推功能的流水线 CPU 电路

```
        end
    end

    fwdb = 2'b00; // default forward b: no hazards
    if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
        fwdb = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
            fwdb = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                fwdb = 2'b11; // select mem_lw
            end
        end
    end
end
```

ALU 的计算结果可以从 EXE 级和 MEM 级前推到 ID 级，而 lw 指令从数据存储器读出的数据只能从 MEM 级前推到 ID 级。这意味着 lw 的后续指令如果与 lw 数据相关，需要把流水线暂停一个周期（“留级”），如图 8.12 所示。

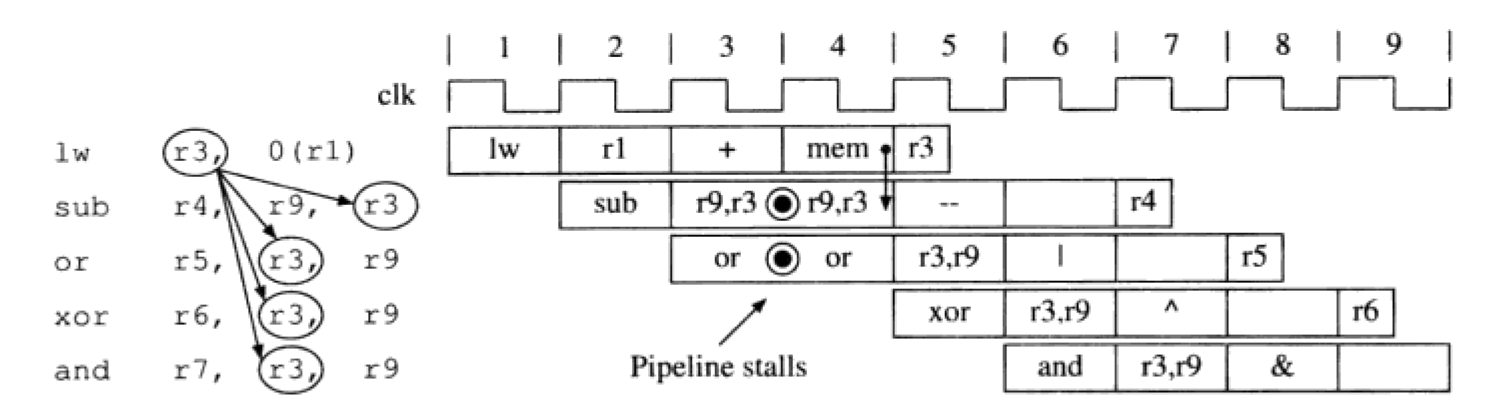


图 8.12 与 lw 指令数据相关需要暂停流水线

我们可以很容易地写出流水线由于 lw 数据相关而需要暂停的条件：

```
stall = ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) |
                                       i_rt & (ern == rt));
```

其中，i_rs 和 i_rt 分别是使用 rs 和 rt 源操作数的指令，ern 是处在 EXE 级的目的寄存器号。如何暂停流水线？非常简单。基本的想法就是不修改 PC 和 IR。我们使用带有使能端的 D 触发器来实现 PC 和 IR。使能端接控制信号 wpcir = ~stall。

图 8.13 带有暂停功能的流水线 CPU 电路

图 8.13 所示的是带有暂停和内部前推功能的流水线 CPU 电路。这里特别需要注意的是，如果不采取其他措施，停一级流水线将导致 IR 中的指令执行两次。因

此我们需要废弃一次指令的执行。如何废弃？非常简单，只需禁止它修改 CPU 的状态 (封锁写信号) 即可。假设原始的写寄存器信号和写存储器信号分别为 wreg_org 和 wmem_org，则新的写寄存器信号 wreg 和写存储器信号 wmem 分别为：

```
wreg = wreg_org & wpcir;
wmem = wmem_org & wpcir;
```

8.2.2 控制相关及解决对策

流水线 CPU 在执行转移或跳转指令时会出现控制相关的问题，即在实际转向目标地址之前，转移或跳转指令的后续指令已经取到流水线中来了，如图 8.14 所示。图 8.14(a) 所示的是转移目标地址及条件在指令 beq 的 EXE 级确定，因而跟在 beq 后面的两条指令已进入流水线。如果转移目标地址及条件能够在 ID 级确定，则仅有一条后续指令进入流水线，如图 8.14(b) 所示。

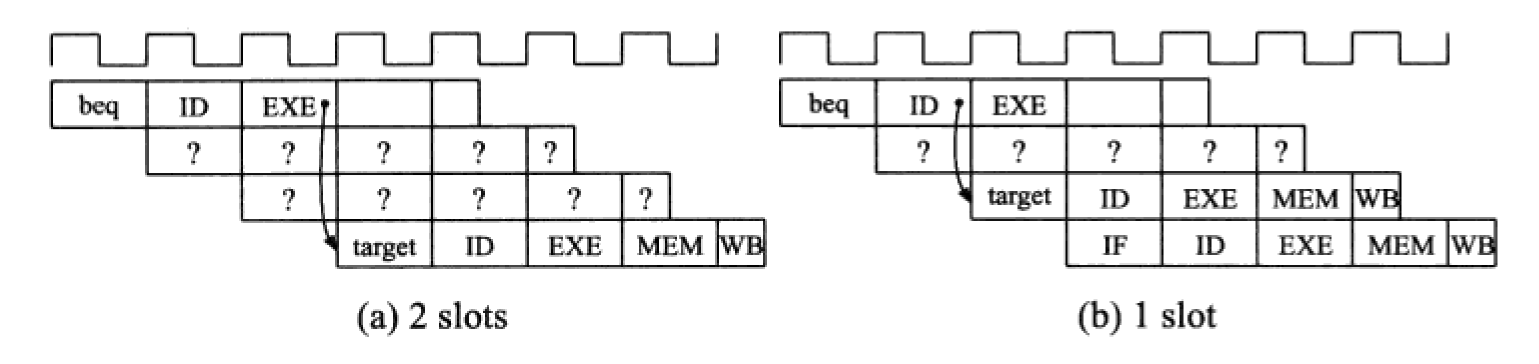


图 8.14 流水线 CPU 的控制相关问题

解决控制相关问题的方法有多种，比如废弃后续指令的执行、延迟转移、转移预测等。我们这里使用一个周期的延迟转移 (Delayed Branch) 技术解决流水线 CPU 的控制相关问题。图 8.15 所示的是使用一个周期的延迟转移技术的流水线 CPU 的时序：不管是否转移，转移指令 (第 i 条) 的后续一条指令 (第 i + 1 条) 总是被执行，就好像它是第 i − 1 条指令。我们称这个位置 (周期) 为延迟槽 (Delay Slot)。现在的任务是确保每条引起转移的指令都能在 ID 级确定是否转移并计算出转移地址。在我们的流水线 CPU 中，引起转移的指令有 5 条：jr、beq、bne、j 和 jal。

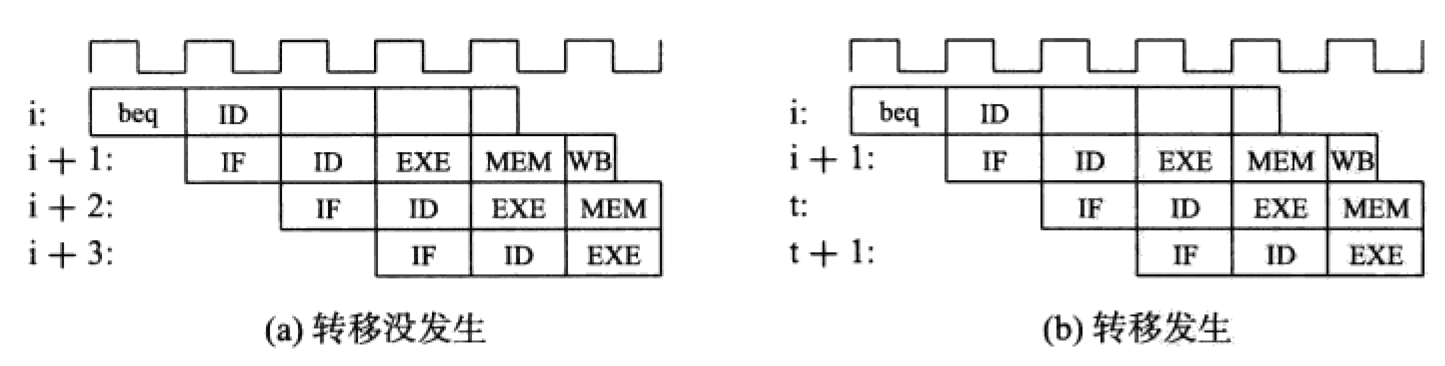


图 8.15 使用一个周期的延迟转移技术的流水线 CPU 的时序

MIPS 指令系统原本就定义这些指令是一个周期的延迟转移指令 (有一个延迟槽)。所有这 5 条指令在 ID 级计算出转移地址是没有问题的。由于 j、jal 和 jr 是无条件转移指令，在 ID 级确定是否转移更是没有问题。有问题的是 beq 和 bne。这是两

条条件转移指令。多周期 CPU 在 EXE 级由 ALU 比较出两个寄存器数据是否相等。为了确保实现一个周期的延迟转移，我们必须在 ID 级比较出两个寄存器数据是否相等。我们可以使用异或门 (XOR) 和或非门 (NOR) 来实现这种比较。

图 8.16 所示的是实现在 ID 级确定是否转移的电路。图中的 equ 比较出两个寄存器数据是否相等，pcsource[1:0] 从 4 个地址中选择一个写入 PC：00 选择 PC + 4 (不转移)，01 选择条件转移的目标地址 (beq 和 bne)，10 选择寄存器数据 (jr)，11 选择跳转指令的目标地址 (j 和 jal)。因此，我们有

```
rsrtequ = ~|(da^db);
pcsource[1] = i_jr | i_j | i_jal;
pcsource[0] = i_beq & rsrtequ | i_bne & ~rsrtequ | i_j | i_jal;
```

其中，da 和 db 是 ID 级的两个新鲜的数据，i_jr、i_j、i_jal、i_beq 和 i_bne 分别是指令 jr、j、jal、beq 和 bne 的译码输出。

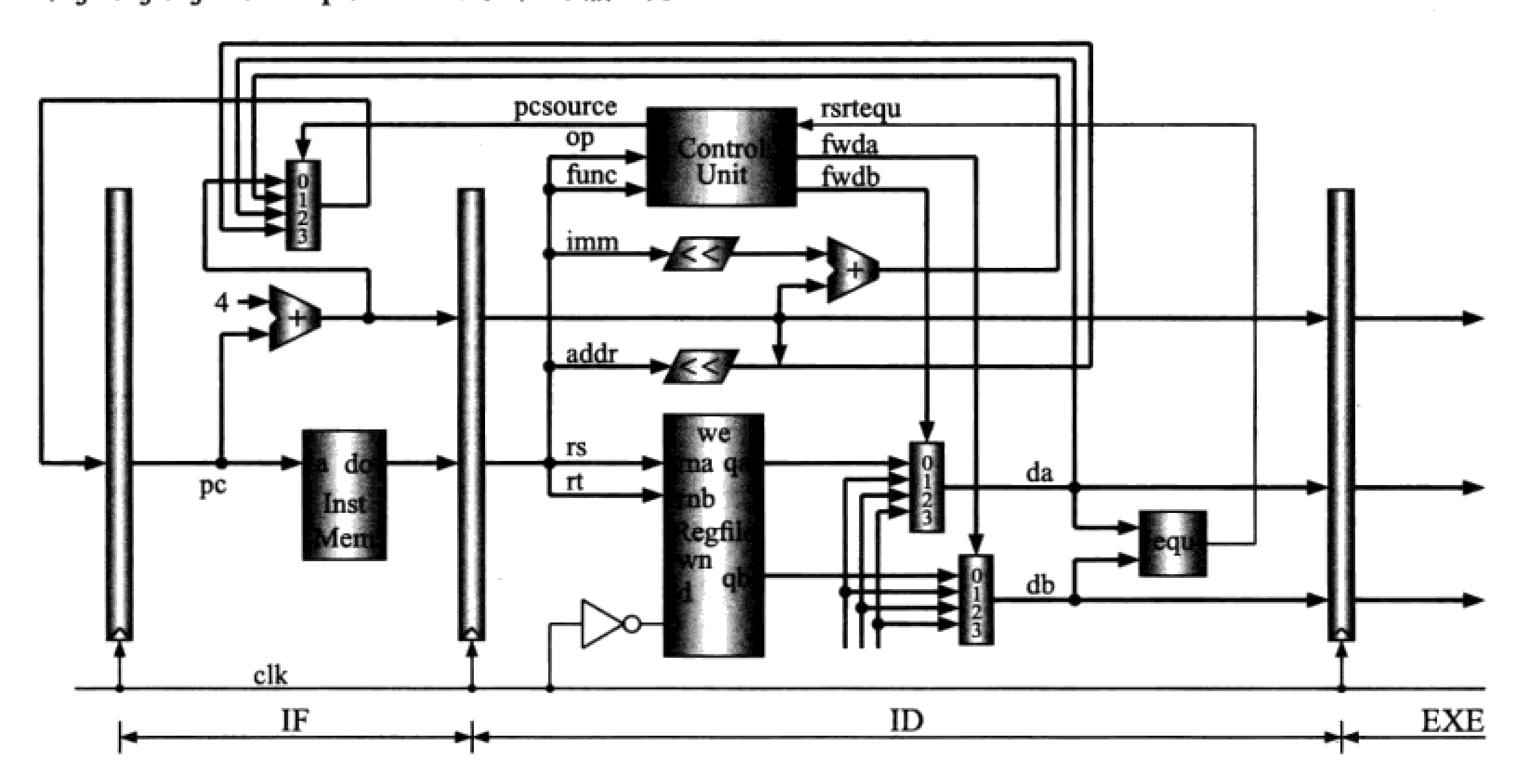


图 8.16 在 ID 级确定是否转移

8.3 流水线 CPU 的整体设计及 Verilog HDL 代码

流水线 CPU 实现的指令与单周期和多周期 CPU 实现的指令相同，见表 4.4。但要注意的是，单周期和多周期 CPU 忽略了 MIPS 转移类指令的延迟转移的特性，我们这里恢复这些指令的本来面目。

8.3.1 流水线 CPU 的整体电路

综合前一节的讨论我们可以画出流水线 CPU 的整体电路，如图 8.17 所示。由于 jal 是一条延迟转移指令，返回地址应为 PC + 8，所以我们在 EXE 级又使用了一个加法器，对 PC + 4 再加 4 (也可在 ID 级实现)。

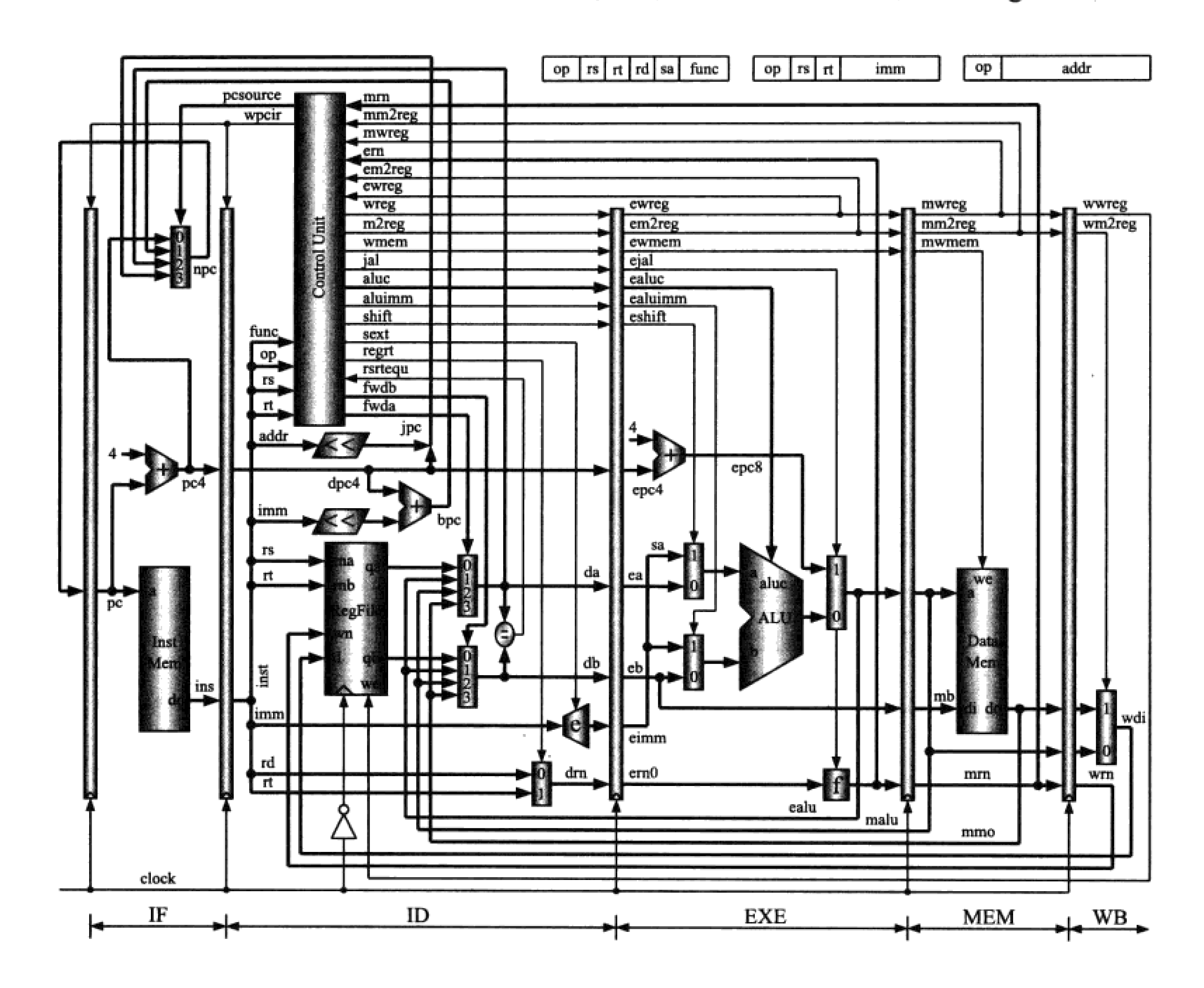


图 8.17 流水线 CPU 的整体电路

8.3.2 流水线 CPU 的 Verilog HDL 代码

本小节给出流水线 CPU 的 Verilog HDL 代码。一些基本模块的代码，比如多路选择器、ALU 和寄存器堆等，已经在前面各章列出了。

1. 最顶层 Verilog HDL 代码

以下是图 8.17 所示的流水线 CPU 整体电路的 Verilog HDL 代码。由以下模块构成：PC、IF 级的组合电路、IF 级与 ID 级之间的流水线寄存器、ID 级的组合电路、ID 级与 EXE 级之间的流水线寄存器、EXE 级的组合电路、EXE 级与 MEM 级之间的流水线寄存器、MEM 级的组合电路、MEM 级与 WB 级之间的流水线寄存器、WB 级的组合电路。

```
module pipelinedcpu (clock,memclock,resetn,pc,inst,ealu,malu,walu);
    input clock,memclock,resetn;
    output [31:0] pc,inst,ealu,malu,walu;
```

```
    wire [31:0] bpc,jpc,npc,pc4,ins,dpc4,inst,da,db,dimm,ea,eb,eimm;
    wire [31:0] epc4,mb,mmo,wmo,wdi;
    wire [4:0]  drn,ern0,ern,mrn,wrn;
    wire [3:0]  daluc,ealuc; // daluc = aluc
    wire [1:0]  pcsource;
    wire        wpcir;
    wire        dwreg,dm2reg,dwmem,daluimm,dshift,djal;
    wire        ewreg,em2reg,ewmem,ealuimm,eshift,ejal;
    wire        mwreg,mm2reg,mwmem;
    wire        wwreg,wm2reg;
    pipepc prog_cnt (npc,wpcir,clock,resetn,pc);
    pipeif if_stage (pcsource,pc,bpc,da,jpc,npc,pc4,ins);
    pipeir inst_reg (pc4,ins,wpcir,clock,resetn,dpc4,inst);
    pipeid id_stage (mwreg,mrn,ern,ewreg,em2reg,mm2reg,dpc4,inst,
                     wrn,wdi,ealu,malu,mmo,wwreg,clock,resetn,
                     bpc,jpc,pcsource,wpcir,dwreg,dm2reg,dwmem,
                     daluc,daluimm,da,db,dimm,drn,dshift,djal);
    pipedereg de_reg (dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,
                      drn, dshift,djal,dpc4,clock,resetn,
                      ewreg,em2reg,ewmem,ealuc,ealuimm,ea,eb,eimm,
                      ern0,eshift,ejal,epc4);
    pipeexe exe_stage (ealuc,ealuimm,ea,eb,eimm,eshift,ern0,epc4,
                       ejal,ern,ealu);
    pipeemreg em_reg (ewreg,em2reg,ewmem,ealu,eb,ern,clock,resetn,
                      mwreg,mm2reg,mwmem,malu,mb,mrn);
    pipemem mem_stage (mwmem,malu,mb,clock,memclock,memclock,mmo);
    pipemwreg mw_reg (mwreg,mm2reg,mmo,malu,mrn,clock,resetn,
                      wwreg,wm2reg,wmo,walu,wrn);
    mux2x32 wb_stage (walu,wmo,wm2reg,wdi);
endmodule
```

2. IF 级 Verilog HDL 代码

1) PC 寄存器

```
module pipepc(npc,wpc,clk,clrn,pc);
    input [31:0] npc;
    input        wpc,clk,clrn;
    output [31:0] pc;
    dffe32 program_counter (npc,clk,clrn,wpc,pc);
endmodule
```

2) IF 级的组合电路 (主要模块是指令存储器)

```
module pipeif (pcsource,pc,bpc,rpc,jpc,npc,pc4,ins);
    input [31:0] pc,bpc,rpc,jpc;
    input [1:0]  pcsource;
```

```
    output [31:0] npc,pc4,ins;
    mux4x32 next_pc (pc4,bpc,rpc,jpc,pcsource,npc);
    cla32 pc_plus4 (pc,32'h4,1'b0,pc4);
    pipeimem inst_mem (pc,ins);
endmodule
```

以下是指令存储器代码。指令存储器初始化文件 pipeimem.mif 在下一节给出。

```
module pipeimem (a,inst);
    input [31:0] a;
    output [31:0] inst;
    lpm_rom  lpm_rom_component (.address(a[7:2]),.q(inst));
    defparam lpm_rom_component.lpm_width    = 32,
             lpm_rom_component.lpm_widthad  = 6,
             lpm_rom_component.lpm_numwords = "unused",
             lpm_rom_component.lpm_file     = "pipeimem.mif",
             lpm_rom_component.lpm_indata   = "unused",
             lpm_rom_component.lpm_outdata  = "unregistered",
             lpm_rom_component.lpm_address_control = "unregistered";
endmodule
```

3. ID 级 Verilog HDL 代码

1) IR 寄存器及 PC + 4 寄存器

```
module pipeir (pc4,ins,wir,clk,clrn,dpc4,inst);
    input [31:0] pc4,ins;
    input        wir,clk,clrn;
    output [31:0] dpc4,inst;
    dffe32 pc_plus4 (pc4,clk,clrn,wir,dpc4);
    dffe32 instruction (ins,clk,clrn,wir,inst);
endmodule
```

2) ID 级组合电路 (主要模块是寄存器堆和控制部件)

```
module pipeid (mwreg,mrn,ern,ewreg,em2reg,mm2reg,dpc4,inst,wrn,
               wdi,ealu,malu,mmo,wwreg,clk,clrn,bpc,jpc,pcsource,
               nostall,wreg,m2reg,wmem,aluc,aluimm,a,b,imm,rn,
               shift,jal);
    input [31:0] dpc4,inst,wdi,ealu,malu,mmo;
    input [4:0]  ern,mrn,wrn;
    input        mwreg,ewreg,em2reg,mm2reg,wwreg;
    input        clk,clrn;
    output [31:0] bpc,jpc,a,b,imm;
    output [4:0]  rn;
    output [3:0]  aluc;
    output [1:0]  pcsource;
```

```
    output          nostall,wreg,m2reg,wmem,aluimm,shift,jal;
    wire [5:0]      op,func;
    wire [4:0]      rs,rt,rd;
    wire [31:0]     qa,qb,br_offset;
    wire [15:0]     ext16;
    wire [1:0]      fwda,fwdb;
    wire            regrt,sext,rsrtequ,e;
    assign          func = inst[5:0];
    assign          op   = inst[31:26];
    assign          rs   = inst[25:21];
    assign          rt   = inst[20:16];
    assign          rd   = inst[15:11];
    assign          jpc  = {dpc4[31:28],inst[25:0],2'b00};
    pipeidcu cu (mwreg,mrn,ern,ewreg,em2reg,mm2reg,rsrtequ,func,
                 op,rs,rt,wreg,m2reg,wmem,aluc,regrt,aluimm,
                 fwda,fwdb,nostall,sext,pcsource,shift,jal);
    regfile rf (rs,rt,wdi,wrn,wwreg,~clk,clrn,qa,qb);
    mux2x5 des_reg_no (rd,rt,regrt,rn);
    mux4x32 alu_a (qa,ealu,malu,mmo,fwda,a);
    mux4x32 alu_b (qb,ealu,malu,mmo,fwdb,b);
    assign          rsrtequ = ~|(a^b); // rsrtequ = (a == b)
    assign          e = sext & inst[15];
    assign          ext16 = {16{e}};
    assign          imm = {ext16,inst[15:0]};
    assign          br_offset = {imm[29:0],2'b00};
    cla32 br_addr (dpc4,br_offset,1'b0,bpc);
endmodule
```

控制部件:

```
module pipeidcu (mwreg,mrn,ern,ewreg,em2reg,mm2reg,rsrtequ,func,op,rs,rt,
                 wreg,m2reg,wmem,aluc,regrt,aluimm,fwda,fwdb,nostall,sext,
                 pcsource,shift,jal);
   input  mwreg, ewreg, em2reg, mm2reg, rsrtequ;
   input [4:0] mrn, ern, rs, rt;
   input [5:0] func, op;
   output      wreg, m2reg,wmem, regrt, aluimm, sext, shift, jal;
   output [3:0] aluc;
   output [1:0] pcsource;
   output [1:0] fwda, fwdb; // forwarding
   output       nostall;    // stall pipeline due to lw dependent
   reg [1:0]    fwda, fwdb;
   wire r_type,i_add,i_sub,i_and,i_or,i_xor,i_sll,i_srl,i_sra,i_jr;
   wire i_addi,i_andi,i_ori,i_xori,i_lw,i_sw,i_beq,i_bne,i_lui,i_j,i_jal;
   and(r_type,~op[5],~op[4],~op[3],~op[2],~op[1],~op[0]);
   and(i_add,r_type, func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
   and(i_sub,r_type, func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
```

```
and(i_and,r_type, func[5],~func[4],~func[3], func[2],~func[1],~func[0]);
and(i_or, r_type, func[5],~func[4],~func[3], func[2],~func[1], func[0]);
and(i_xor,r_type, func[5],~func[4],~func[3], func[2], func[1],~func[0]);
and(i_sll,r_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_srl,r_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_sra,r_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_jr, r_type,~func[5],~func[4], func[3],~func[2],~func[1],~func[0]);
and(i_addi,~op[5],~op[4], op[3],~op[2],~op[1],~op[0]);
and(i_andi,~op[5],~op[4], op[3], op[2],~op[1],~op[0]);
and(i_ori, ~op[5],~op[4], op[3], op[2],~op[1], op[0]);
and(i_xori,~op[5],~op[4], op[3], op[2], op[1],~op[0]);
and(i_lw,   op[5],~op[4],~op[3],~op[2], op[1], op[0]);
and(i_sw,   op[5],~op[4], op[3],~op[2], op[1], op[0]);
and(i_beq, ~op[5],~op[4],~op[3], op[2],~op[1],~op[0]);
and(i_bne, ~op[5],~op[4],~op[3], op[2],~op[1], op[0]);
and(i_lui, ~op[5],~op[4], op[3], op[2], op[1], op[0]);
and(i_j,   ~op[5],~op[4],~op[3],~op[2], op[1],~op[0]);
and(i_jal, ~op[5],~op[4],~op[3],~op[2], op[1], op[0]);
wire i_rs = i_add  | i_sub | i_and  | i_or  | i_xor | i_jr  | i_addi |
            i_andi | i_ori | i_xori | i_lw  | i_sw  | i_beq | i_bne;
wire i_rt = i_add  | i_sub | i_and  | i_or  | i_xor | i_sll | i_srl  |
            i_sra  | i_sw  | i_beq  | i_bne;
assign nostall = ~(ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) |
                                                  i_rt & (ern == rt)));
always @ (ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or
          rt) begin
    fwda = 2'b00; // default forward a: no hazards
    if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
        fwda = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
            fwda = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                fwda = 2'b11; // select mem_lw
            end
        end
    end
    fwdb = 2'b00; // default forward b: no hazards
    if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
        fwdb = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
            fwdb = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                fwdb = 2'b11; // select mem_lw
```

```
              end
          end
      end
  end
  assign wreg    = (i_add  | i_sub  | i_and  | i_or   | i_xor | i_sll  |
                    i_srl  | i_sra  | i_addi | i_andi | i_ori | i_xori |
                    i_lw   | i_lui  | i_jal) & nostall;
  assign regrt    = i_addi | i_andi | i_ori  | i_xori | i_lw  | i_lui;
  assign jal      = i_jal;
  assign m2reg    = i_lw;
  assign shift    = i_sll  | i_srl  | i_sra;
  assign aluimm   = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui | i_sw;
  assign sext     = i_addi | i_lw   | i_sw  | i_beq  | i_bne;
  assign aluc[3] = i_sra;
  assign aluc[2] = i_sub  | i_or   | i_srl | i_sra  | i_ori  | i_lui;
  assign aluc[1] = i_xor  | i_sll  | i_srl | i_sra  | i_xori | i_beq  |
                    i_bne  | i_lui;
  assign aluc[0] = i_and  | i_or   | i_sll | i_srl  | i_sra  | i_andi |
                    i_ori;
  assign wmem     = i_sw & nostall;
  assign pcsource[1] = i_jr | i_j | i_jal;
  assign pcsource[0] = i_beq & rsrtequ | i_bne & ~rsrtequ | i_j | i_jal;
endmodule
```

4. EXE 级 Verilog HDL 代码

1) ID 级和 EXE 级之间的流水线寄存器

```
module pipedereg (dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,drn,
                  dshift,djal,dpc4,clk,clrn,ewreg,em2reg,ewmem,
                  ealuc,ealuimm,ea,eb,eimm,ern,eshift,ejal,epc4);
    input  [31:0] da,db,dimm,dpc4;
    input  [4:0]  drn;
    input  [3:0]  daluc;
    input         dwreg,dm2reg,dwmem,daluimm,dshift,djal;
    input         clk,clrn;
    output  [31:0] ea,eb,eimm,epc4;
    output  [4:0]  ern;
    output  [3:0]  ealuc;
    output         ewreg,em2reg,ewmem,ealuimm,eshift,ejal;
    reg  [31:0]    ea,eb,eimm,epc4;
    reg  [4:0]     ern;
    reg  [3:0]     ealuc;
    reg            ewreg,em2reg,ewmem,ealuimm,eshift,ejal;
    always @(negedge clrn or posedge clk)
      if (clrn == 0) begin
          ewreg   <= 0;          em2reg   <= 0;
```

```
            ewmem   <= 0;           ealuc   <= 0;
            ealuimm <= 0;           ea      <= 0;
            eb      <= 0;           eimm    <= 0;
            ern     <= 0;           eshift  <= 0;
            ejal    <= 0;           epc4    <= 0;
        end else begin
            ewreg   <= dwreg;       em2reg  <= dm2reg;
            ewmem   <= dwmem;       ealuc   <= daluc;
            ealuimm <= daluimm;     ea      <= da;
            eb      <= db;          eimm    <= dimm;
            ern     <= drn;         eshift  <= dshift;
            ejal    <= djal;        epc4    <= dpc4;
        end
endmodule
```

2) EXE 级的组合电路 (主要模块是 ALU 和返回地址的产生)

```
module pipeexe (ealuc,ealuimm,ea,eb,eimm,eshift,ern0,epc4,ejal,ern,
                ealu);
    input [31:0]  ea,eb,eimm,epc4;
    input [4:0]   ern0;
    input [3:0]   ealuc;
    input         ealuimm,eshift,ejal;
    output [31:0] ealu;
    output [4:0]  ern;
    wire [31:0]   alua,alub,sa,ealu0,epc8;
    wire          z;
    assign        sa = {eimm[5:0],eimm[31:6]}; // shift amount
    cla32 ret_addr (epc4,32'h4,1'b0,epc8);
    mux2x32 alu_ina (ea,sa,eshift,alua);
    mux2x32 alu_inb (eb,eimm,ealuimm,alub);
    mux2x32 save_pc8 (ealu0,epc8,ejal,ealu);
    assign        ern = ern0 | {5{ejal}};
    alu al_unit (alua,alub,ealuc,ealu0,z);
endmodule
```

5. MEM 级 Verilog HDL 代码

1) EXE 级和 MEM 级之间的流水线寄存器

```
module pipeemreg (ewreg,em2reg,ewmem,ealu,eb,ern,clk,clrn,
                  mwreg,mm2reg,mwmem,malu,mb,mrn);
    input [31:0] ealu,eb;
    input [4:0]  ern;
    input        ewreg,em2reg,ewmem;
    input        clk,clrn;
    output [31:0] malu,mb;
```

```
    output [4:0]  mrn;
    output        mwreg,mm2reg,mwmem;
    reg [31:0]    malu,mb;
    reg [4:0]     mrn;
    reg           mwreg,mm2reg,mwmem;
    always @(negedge clrn or posedge clk)
      if (clrn == 0) begin
          mwreg  <= 0;        mm2reg <= 0;
          mwmem  <= 0;        malu   <= 0;
          mb     <= 0;        mrn    <= 0;
      end else begin
          mwreg  <= ewreg;    mm2reg <= em2reg;
          mwmem  <= ewmem;    malu   <= ealu;
          mb     <= eb;       mrn    <= ern;
      end
endmodule
```

2) 数据存储器代码。数据存储器初始化文件 pipedmem.mif 在下一节给出。

```
module pipemem (we,addr,datain,clk,inclk,outclk,dataout);
    input  [31:0] addr,datain;
    input         clk,we,inclk,outclk;
    output [31:0] dataout;

    wire          write_enable = we & ~clk;
    lpm_ram_dq ram (.data(datain),.address(addr[6:2]),
                    .we(write_enable),.inclock(inclk),
                    .outclock(outclk),.q(dataout));
    defparam   ram.lpm_width    = 32;
    defparam   ram.lpm_widthad  =  5;
    defparam   ram.lpm_indata   = "registered";
    defparam   ram.lpm_outdata  = "registered";
    defparam   ram.lpm_file     = "pipedmem.mif";
    defparam   ram.lpm_address_control = "registered";
endmodule
```

6. WB 级 Verilog HDL 代码

1) MEM 级和 WB 级之间的流水线寄存器

```
module pipemwreg (mwreg,mm2reg,mmo,malu,mrn,clk,clrn,
                  wwreg,wm2reg,wmo,walu,wrn);
    input [31:0] mmo,malu;
    input [4:0]  mrn;
    input        mwreg,mm2reg;
    input        clk,clrn;
```

```
    output [31:0] wmo,walu;
    output [4:0]  wrn;
    output        wwreg,wm2reg;
    reg [31:0]    wmo,walu;
    reg [4:0]     wrn;
    reg           wwreg,wm2reg;
    always @(negedge clrn or posedge clk)
      if (clrn == 0) begin
          wwreg  <= 0;       wm2reg <= 0;       wmo <= 0;
          walu   <= 0;       wrn    <= 0;
      end else begin
          wwreg  <= mwreg;   wm2reg <= mm2reg;  wmo <= mmo;
          walu   <= malu;    wrn    <= mrn;
      end
endmodule
```

2) WB 级的组合电路只有一个多路选择器，已在最顶层模块中列出。

8.4 流水线 CPU 的测试

8.4.1 流水线 CPU 的测试程序

指令存储器初始化文件 pipeimem.mif 如下所示。注意，该程序不是最优化的，比如要测试 lw 指令在数据相关时暂停流水线的情况，我们故意在它的下面安排了一条 sub 指令，使用 lw 指令从存储器取来的数据。

```
DEPTH = 64;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
[0..1F] : 00000000; % Range--Every address from 0 to 1F = 00000000         %
 0 : 3c010000; % (00) main:   lui  r1, 0          # address of data[0]     %
 1 : 34240050; % (04)         ori  r4, r1, 80     # address of data[0]     %
 2 : 0c00001b; % (08) call:   jal  sum            # call function          %
 3 : 20050004; % (0c) dslot1: addi r5, r0,  4     # counter,DELYED SLOT(DS)%
 4 : ac820000; % (10) return: sw   r2, 0(r4)      # store result           %
 5 : 8c890000; % (14)         lw   r9, 0(r4)      # check sw               %
 6 : 01244022; % (18)         sub  r8, r9, r4     # sub: r8 <-- r9 - r4    %
 7 : 20050003; % (1c)         addi r5, r0,  3     # counter                %
 8 : 20a5ffff; % (20) loop2:  addi r5, r5, -1     # counter - 1            %
 9 : 34a8ffff; % (24)         ori  r8, r5, 0xffff# zero-extend: 0000ffff  %
 A : 39085555; % (28)         xori r8, r8, 0x5555# zero-extend: 0000aaaa  %
 B : 2009ffff; % (2c)         addi r9, r0, -1     # sign-extend: ffffffff  %
```

```
 C : 312affff; % (30)         andi r10, r9,0xffff# zero-extend: 0000ffff  %
 D : 01493025; % (34)         or   r6, r10, r9   # or:  ffffffff          %
 E : 01494026; % (38)         xor  r8, r10, r9   # xor: ffff0000          %
 F : 01463824; % (3c)         and  r7, r10, r6   # and: 0000ffff          %
10 : 10a00003; % (40)         beq  r5, r0, shift # if r5 = 0, goto shift  %
11 : 00000000; % (44) dslot2: nop                # DS                     %
12 : 08000008; % (48)         j    loop2         # jump loop2             %
13 : 00000000; % (4c) dslot3: nop                # DS                     %
14 : 2005ffff; % (50) shift:  addi r5, r0, -1    # r5   = ffffffff        %
15 : 000543c0; % (54)         sll  r8, r5, 15    # <<15 = ffff8000        %
16 : 00084400; % (58)         sll  r8, r8, 16    # <<16 = 80000000        %
17 : 00084403; % (5c)         sra  r8, r8, 16    # >>16 = ffff8000(arith) %
18 : 000843c2; % (60)         srl  r8, r8, 15    # >>15 = 0001ffff(logic) %
19 : 08000019; % (64) finish: j    finish        # dead loop              %
1A : 00000000; % (68) dslot4: nop                # DS                     %
1B : 00004020; % (6c) sum:    add  r8, r0, r0    # sum                    %
1C : 8c890000; % (70) loop:   lw   r9, 0(r4)     # load data              %
1D : 01094020; % (74)         add  r8, r8, r9    # sum                    %
1E : 20a5ffff; % (78)         addi r5, r5, -1    # counter - 1            %
1F : 14a0fffc; % (7c)         bne  r5, r0, loop  # finish?                %
20 : 20840004; % (80) dslot5: addi r4, r4,  4    # address + 4, DS        %
21 : 03e00008; % (84)         jr   r31           # return                 %
22 : 00081000; % (88) dslot6: sll  r2, r8, 0     # move result to v0, DS  %
END ;
```

数据存储器初始化文件 pipedmem.mif:

```
DEPTH = 32;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %

ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
[0..1F] : 00000000; % Range--Every address from 0 to 1F = 0    %
   0 : BF800000; % 1 01111111 00..0 fp -1                      %
  14 : 000000A3; % (50) data[0]   0 +  A3 =  A3                %
  15 : 00000027; % (54) data[1]  A3 +  27 =  CA                %
  16 : 00000079; % (58) data[2]  CA +  79 = 143                %
  17 : 00000115; % (5C) data[3] 143 + 115 = 258                %
END ;
```

8.4.2 流水线 CPU 的仿真波形

以下波形图是流水线 CPU 执行以上程序时 Quartus II 的输出结果，其中的信号按流水线级 IF、ID、EXE、MEM 和 WB 次序排列，每级一个。

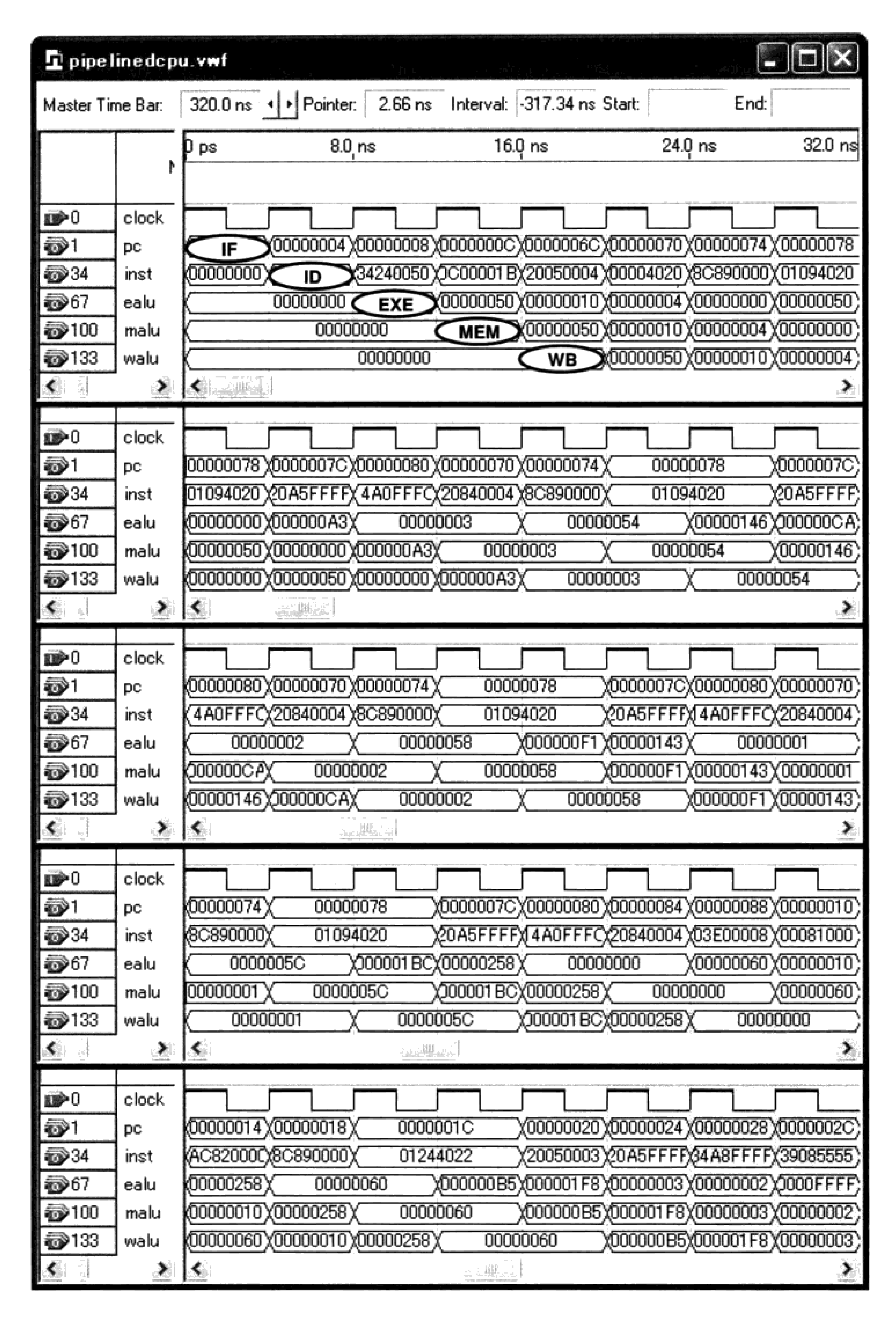


图 8.18　流水线 CPU 仿真波形图 (1 ～ 5)

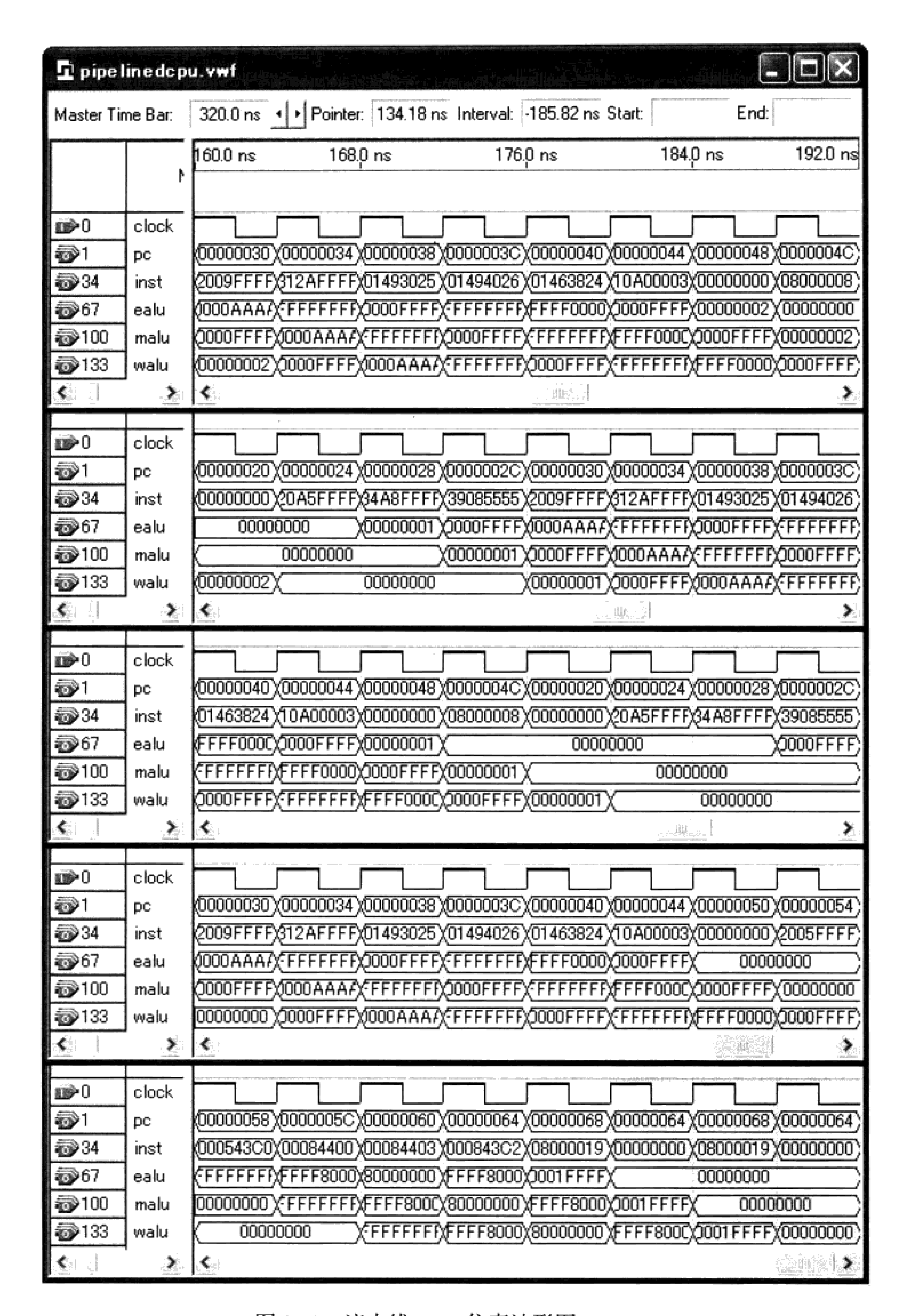


图 8.19 流水线 CPU 仿真波形图 (6 ~ 10)

8.5 精确中断和异常事件处理

在第 6 章，我们描述了单周期 CPU 的异常和中断处理，本节描述流水线 CPU 如何处理异常和中断。与单周期和多周期 CPU 不同，流水线 CPU 有多条指令同时在运行，你找不出一个时间点，说所有在流水线的指令都执行完毕。因此，流水线 CPU 处理异常和中断要比单周期和多周期 CPU 处理异常和中断复杂得多。另外流水线 CPU 还允许延迟转移，进一步增加了流水线 CPU 处理异常和中断的复杂程度。

如果流水线 CPU 处理异常和中断能像单周期 CPU 做得那样漂亮，我们就说该流水线 CPU 具有精确中断 (Precise Interrupt/Exception) 机制。具体地讲，如果我们把中断或异常发生的时刻定为一个时间点，那么精确中断就是要保证在该时间点以前的指令都要安全地执行完，在该时间点以后的指令就像没执行过似的，即不改变计算机的任何状态。本节讲述在流水线 CPU 中如何实现精确中断并给出具体的 Verilog HDL 代码。

8.5.1 异常事件和中断的种类以及相关的寄存器

流水线 CPU 处理异常事件和中断时所使用的寄存器见图 8.20。它们与图 6.9 给出的 3 个寄存器基本相同，只是在 Cause 寄存器中增加了一位 BD。如果引起异常事件的指令是在转移或跳转指令的延迟槽中，BD 置 1，正常情况下清零。如果有外部中断请求并且这时在流水线 ID 级的指令是处在延迟槽中，也把 BD 置 1。

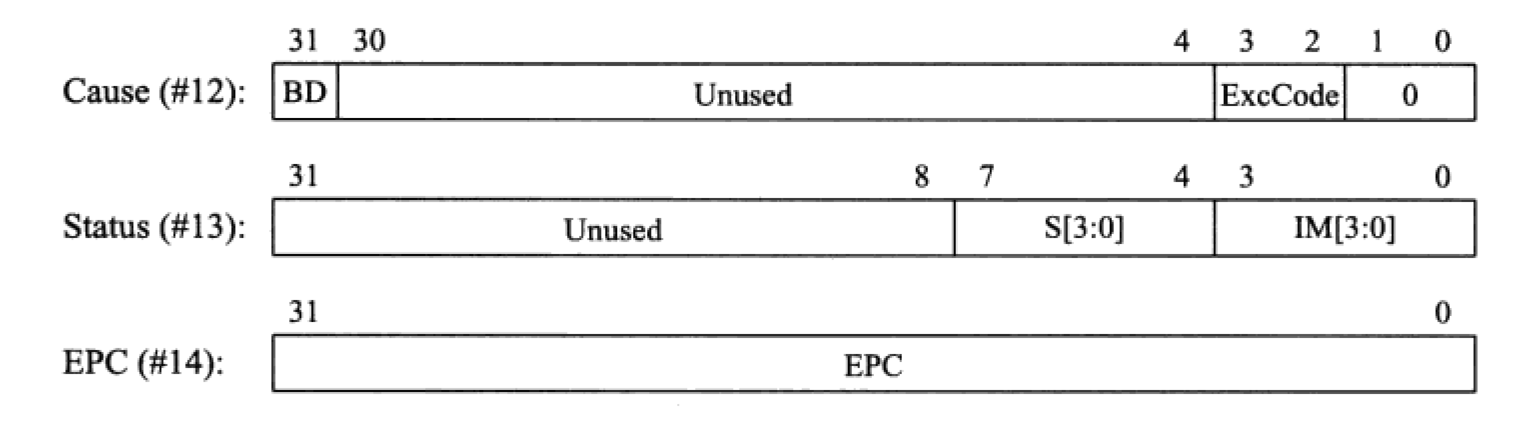


图 8.20 与流水线 CPU 处理异常事件和中断有关的 3 个寄存器

我们仍以第 6 章描述的中断和异常的种类为例加以说明，见表 8.1 和图 8.21。注意表中最右列给出了异常或中断可能出现在流水线的哪一级。

表 8.1 各流水线级可能出现的异常或中断

ExcCode	助记符	种类	屏蔽	描 述	出现在
0	Int	中断	IM[0]	外部中断	任意级
1	Sys	异常	IM[1]	执行系统调用指令	ID 级
2	Unimpl	异常	IM[2]	试图执行没有实现的指令	ID 级
3	Ov	异常	IM[3]	算术操作时结果溢出	EXE 级

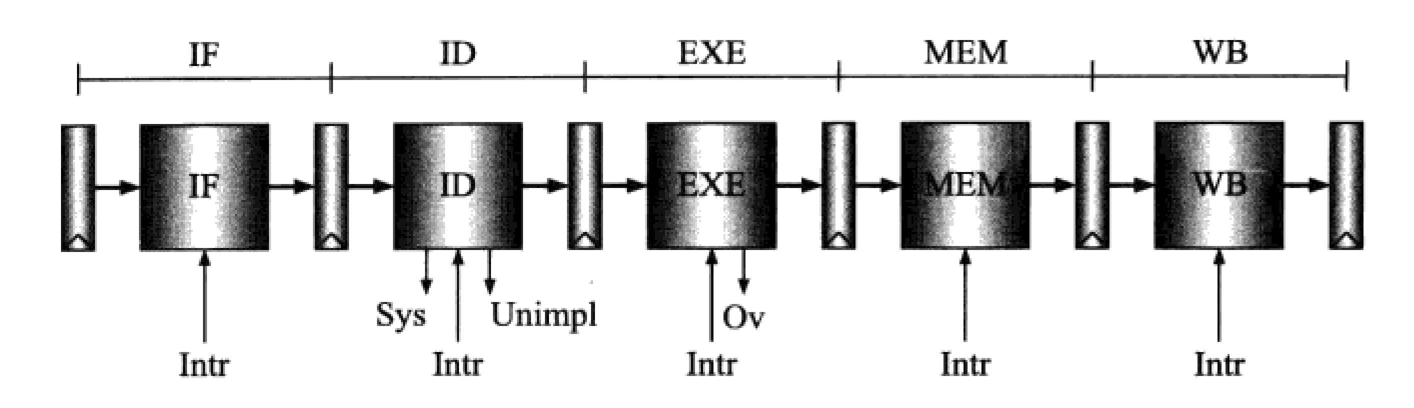


图 8.21 各流水线级可能出现的异常或中断

按照老规矩，中断时 EPC 保存返回地址；异常时 EPC 保存引起异常的指令的地址。但是，如果引起异常的指令是在延迟槽中，则 EPC 保存延迟转移指令的地址。因此我们必须要用某种手段判断出一条指令是否是在延迟槽中。还有，当异常或中断发生时，我们需要报废后续指令甚至当前指令。

见图 8.22，假设指令 I2 是引起异常的指令。如果异常事件是 Sys 或 Unimpl，则在 ID 级就可判断出来。这时，可以生成 cancel 信号，经流水线寄存器送到 EXE 级 (e_cancel) 去封锁下一条指令 (I3) 在 ID 级的信号。如果要封锁下下一条指令 (I4)，可使用 MEM 级的 m_cancel。

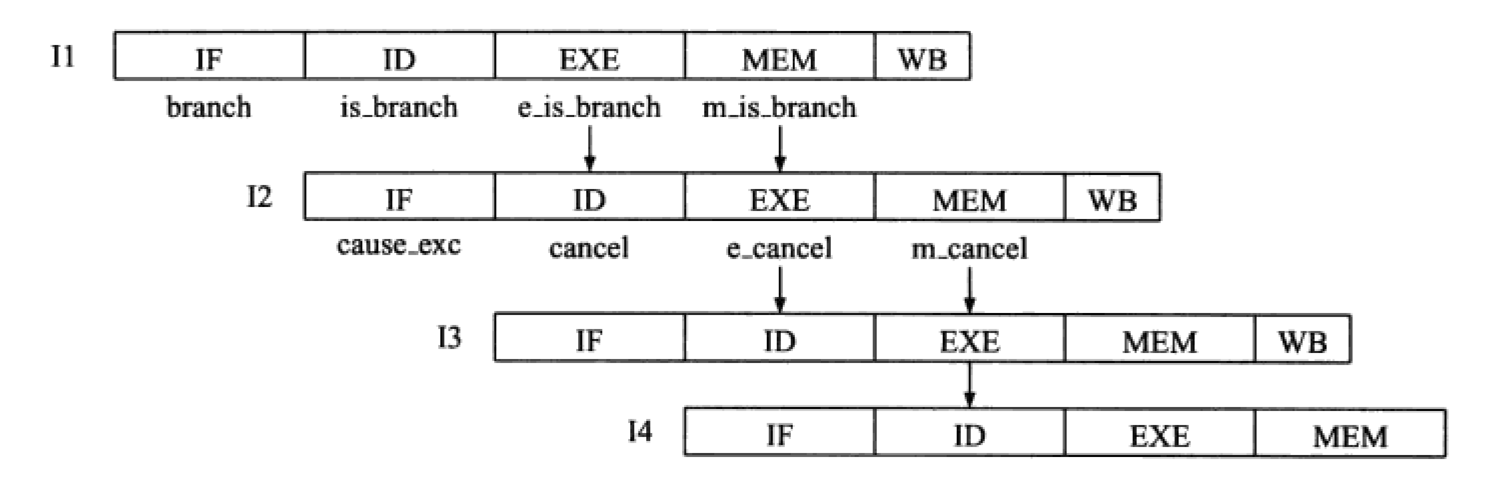


图 8.22 判断延迟槽及报废指令的手段

至于要确认 ID 级的指令是否处在延迟槽，则可使用 EXE 级的 e_is_branch 信号，它来自于 ID 级对指令的译码。如果是转移或跳转指令，则将 is_branch 置 1。同理，使用 m_is_branch 可以确定 EXE 级的指令是否在延迟槽中。

8.5.2 流水线 CPU 的中断响应过程

CPU 响应中断时要停止当前程序的执行，转入中断处理程序，处理完毕再返回。如果中断发生时 CPU 正在执行转移指令或正在执行处在延迟槽中的指令，情况就会变得比较复杂。一种解决方案是把中断请求先保存下来，等到 CPU 比较“稳定”时再响应中断。这就相当于单位的领导说“知道了，等我们研究研究”。本书的做法不是这样，不管领导有多忙，中断请求来了就立即响应。

为了实现精确中断，我们把中断请求出现的时刻分为以下三种情况：

1) 当中断请求出现时，处在 ID 级的指令刚好是一条转移 (或跳转) 指令；
2) 当中断请求出现时，ID 级的指令刚好处在延迟槽；
3) 除了情况 1 和 2 的其他情况 (一般情况)。

1. ID 级执行转移指令时来了中断

转移指令要把转移的目标地址送入 PC，而响应中断时要把中断处理程序的入口地址送入 PC。两个地址只能有一个送入 PC。送谁？当然是中断处理程序的入口地址了。转移指令的武功就这样被废了。但不用伤心，我们把转移指令的地址送入 EPC (注意不是转移的目标地址)。从中断处理程序返回时，再重新执行它。

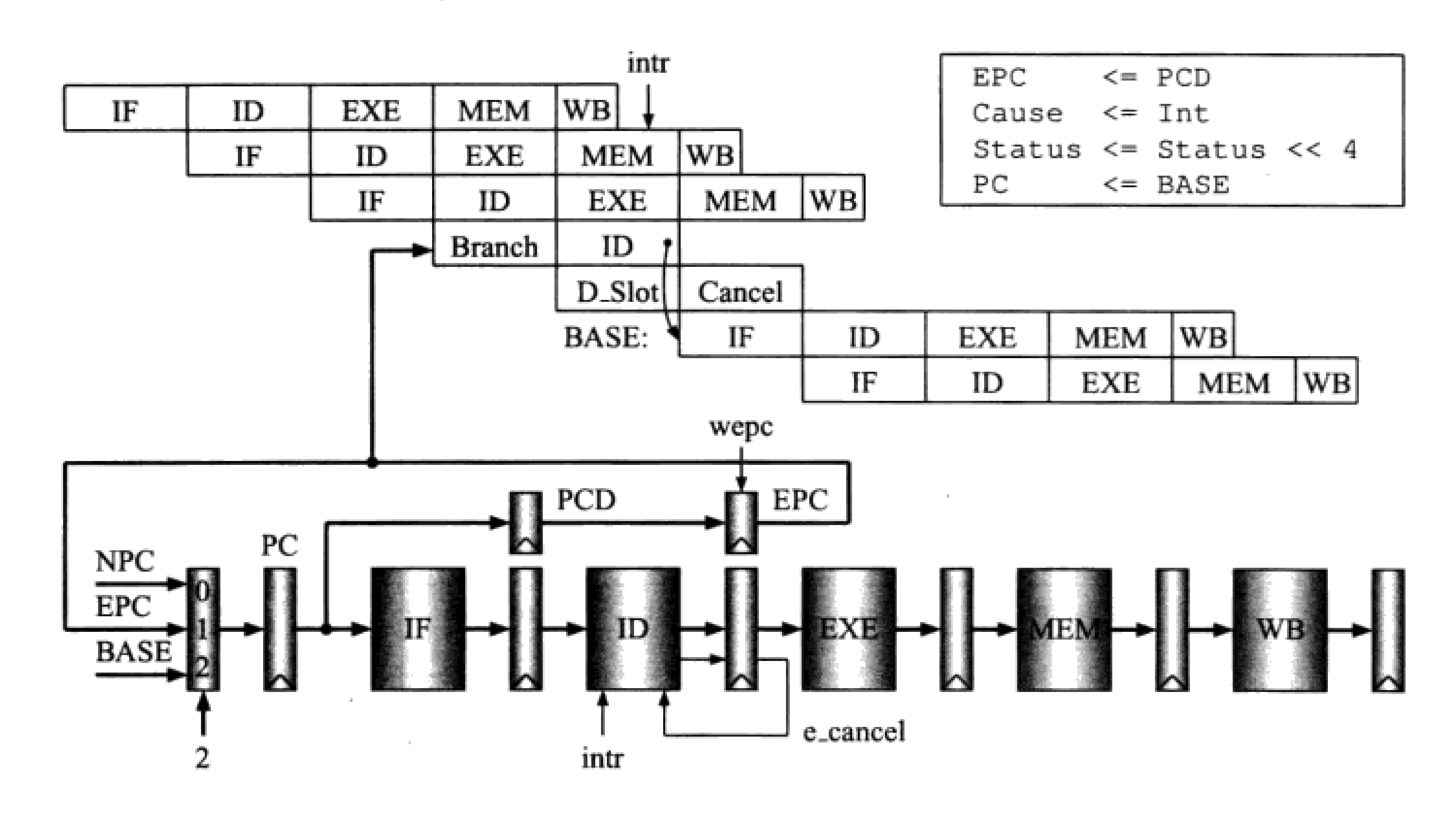


图 8.23 在转移指令的 ID 级遇见中断

见图 8.23。如果转移指令的地址是 IF 级的 PC，那么在 ID 级就是 PCD (跟着指令走)。ID 级结束时，把 PCD 写入 EPC (返回地址)，把异常和中断处理程序的入口地址 BASE 写入 PC。与此同时，在 ID 级产生的 cancel 信号也被写入流水线寄存器，它在 EXE 级的输出为 e_cancel，用来废掉下一条 (处在延迟槽的) 指令的武功。从中断返回时，直接把 EPC 的值写入 PC，重新执行转移指令。

2. ID 级是延迟槽时来了中断

中断来时 ID 级正好是延迟槽怎么办？让延迟槽的指令执行完。那么往 EPC 写什么返回地址呢？当然应该是延迟槽上面的转移指令计算出的转移目标地址了，也就是现在正忙着取指令的那个 PC。注意 Cause 寄存器中的 BD 位要置 1，见图 8.24。

图 8.24 中在延迟槽的指令的 ID 级结束时，转移目标地址的指令实际上已经取来了，我们要用 e_cancel 把它废掉。从中断返回时，要重新执行这条被废掉的指令。因

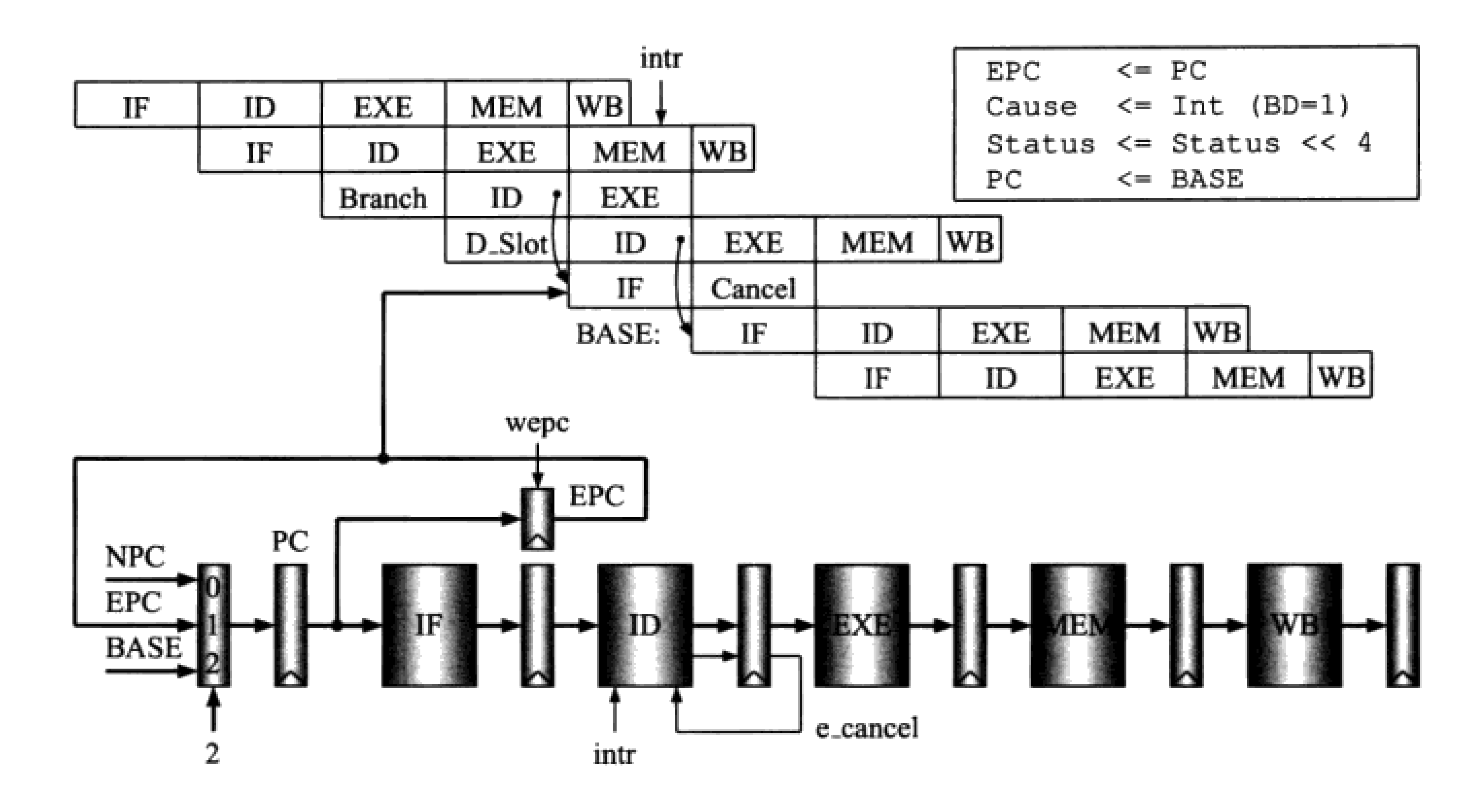


图 8.24 在延迟槽的指令的 ID 级遇见中断

此要把当前的 PC 保存在 EPC。转移指令要把自己是转移指令的消息送到 EXE 级，以使 ID 级的指令能够判断出自己是否在延迟槽中过日子呢。

3. 一般情况下来了中断

一般情况下 (ID 级的指令不是转移指令并且 ID 级也不处在延迟槽) 来了中断比较好办：在 ID 级响应中断，废弃下一条指令，并把下一条指令的地址 (也就是 PC) 写入 EPC，见图 8.25。它与第 2 种情况类似，只是不需要设置 BD。

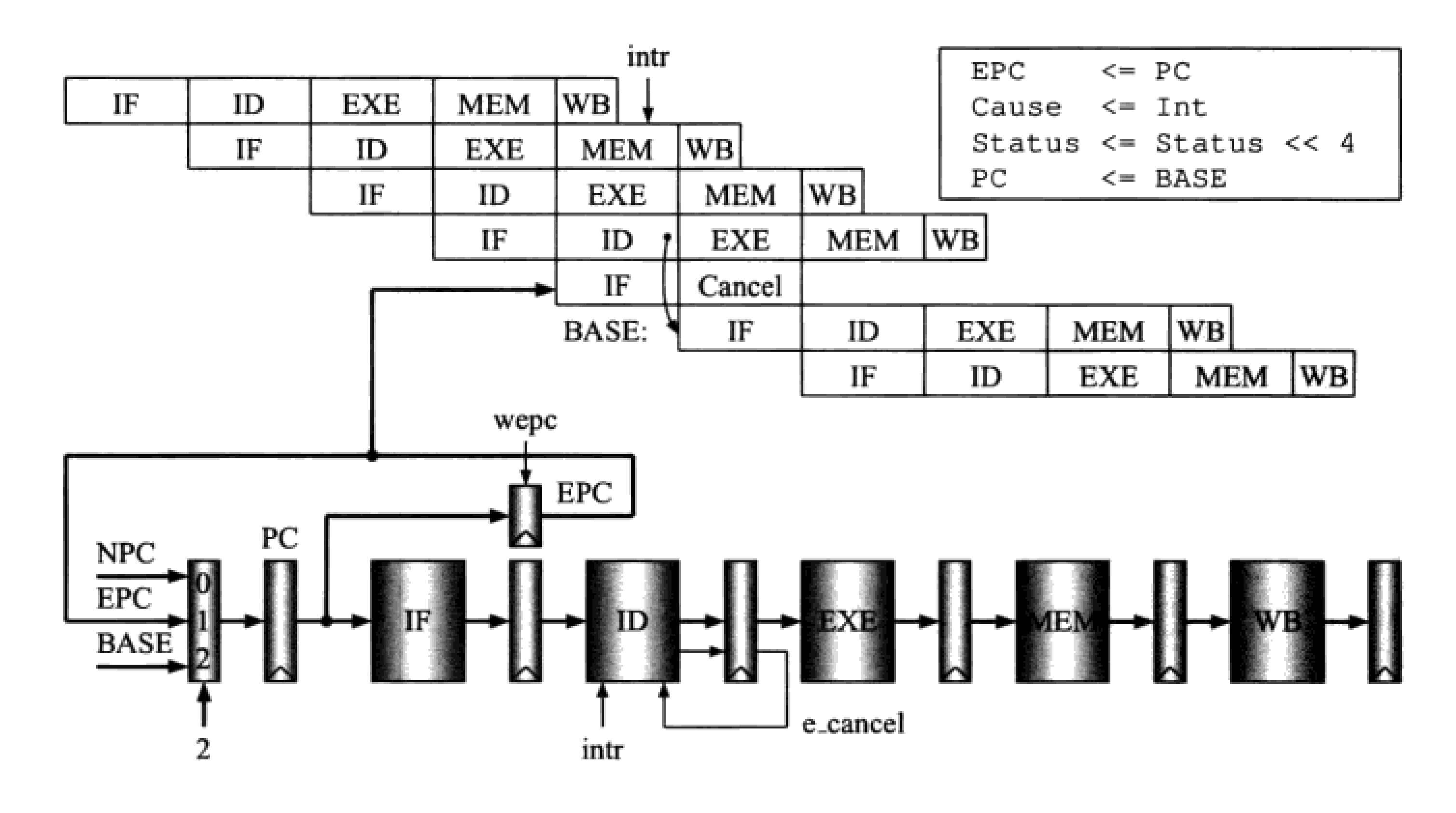


图 8.25 一般情况下遇见中断

8.5.3 流水线 CPU 处理异常事件

流水线 CPU 处理异常事件时是把引起异常事件的指令的地址写入 EPC。如果该指令是处在延迟槽中，则把它上面的转移或跳转指令的地址写入 EPC，并把 BD 位置 1。以下详细描述流水线 CPU 处理系统调用 syscall、未实现的指令 unimpl 和算术结果溢出三种异常事件时所采取的措施。

1. 系统调用 syscall

无论是使用汇编语言编程还是使用高级语言编程然后编译，我们都可以不让系统调用指令出现在延迟槽中，因此我们只考虑通常情况下系统调用指令的执行情况。图 8.26 是流水线 CPU 执行系统调用 syscall 指令时的流水状况：在 ID 级转到异常和中断处理程序并废弃它下面的指令。EPC 保存的是 syscall 指令的地址 PCD。图中 EPC 输入端接一多路器，修改 EPC 时选择 DATA。

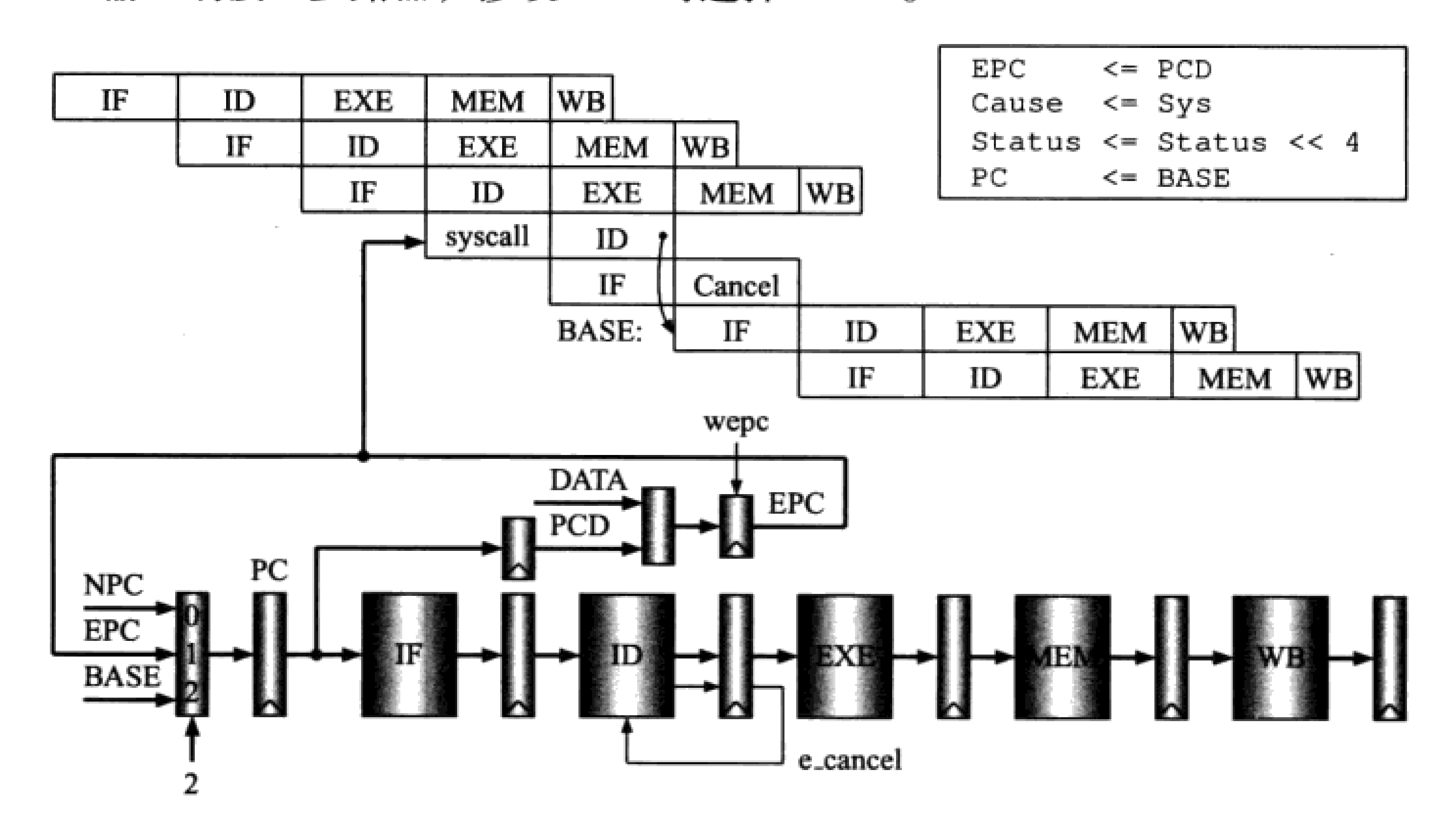


图 8.26 执行 syscall 指令时的流水线

2. 未实现的指令 unimpl

图 8.27 是流水线 CPU 执行处在延迟槽的未实现的指令时的流水线。此时 EPC 要保存的是它上面的转移指令的地址 PCE，Cause 寄存器中的 BD 位要置 1。

图 8.28 是流水线 CPU 执行一般情况下的未实现的指令时的流水线。它与执行 syscall 指令时的流水线类似。

3. 算术结果溢出

结果溢出出现在 EXE 级，比人家要晚一个周期。注意溢出时结果不能往寄存器堆中保存，因此要在 EXE 级封锁 wreg 信号。另外，ID 级的指令也要被废弃。

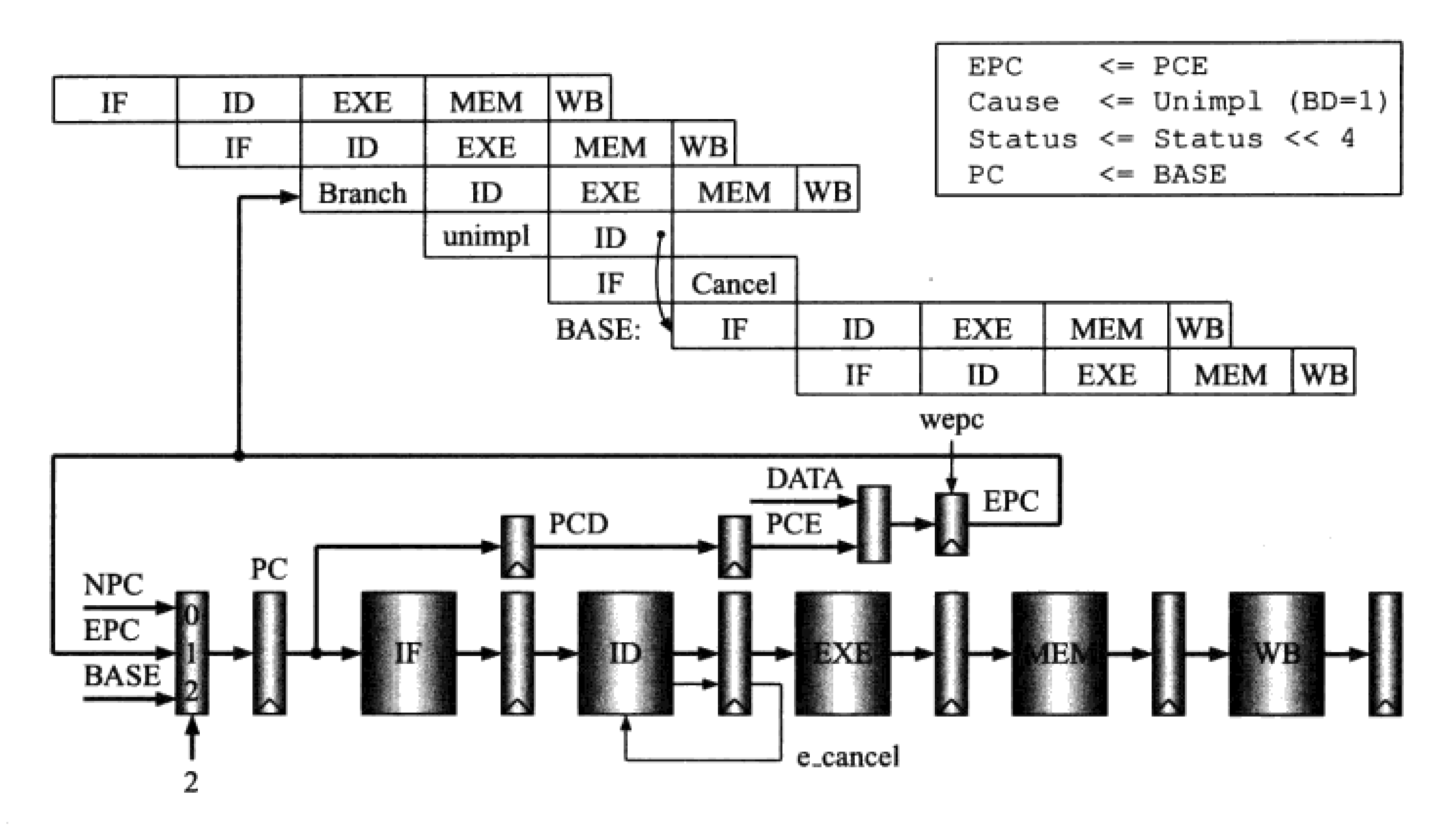


图 8.27 执行处在延迟槽的未实现的指令时的流水线

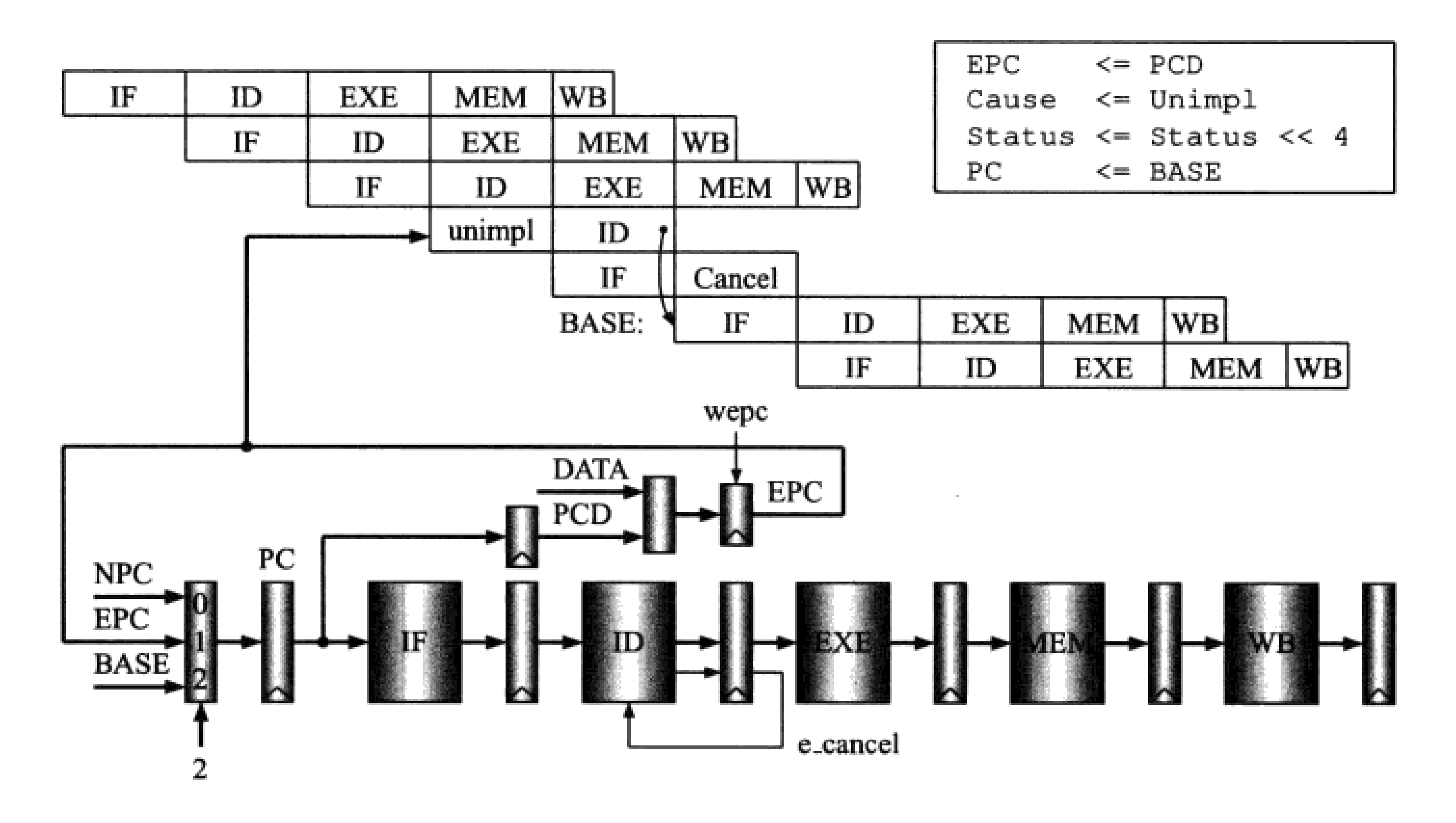


图 8.28 执行一般情况下的未实现的指令时的流水线

图 8.29 是流水线 CPU 处在延迟槽的指令结果溢出时的流水线。此时 EPC 要保存的是它上面的转移指令的地址 PCM。Cause 寄存器中的 BD 位要置 1。

图 8.30 是流水线 CPU 一般情况下的结果溢出时的流水线。EPC 要保存的是引起溢出指令的地址 PCE。

表 8.2 总结了异常事件和中断出现时 CPU 在 ID 级结束时往 EPC 中写入的内容。Sys 和 Unimpl 是在 ID 级被判断出，它们都不可能是转移指令。转移指令本身在 ID 级不使用 ALU，不会上溢。

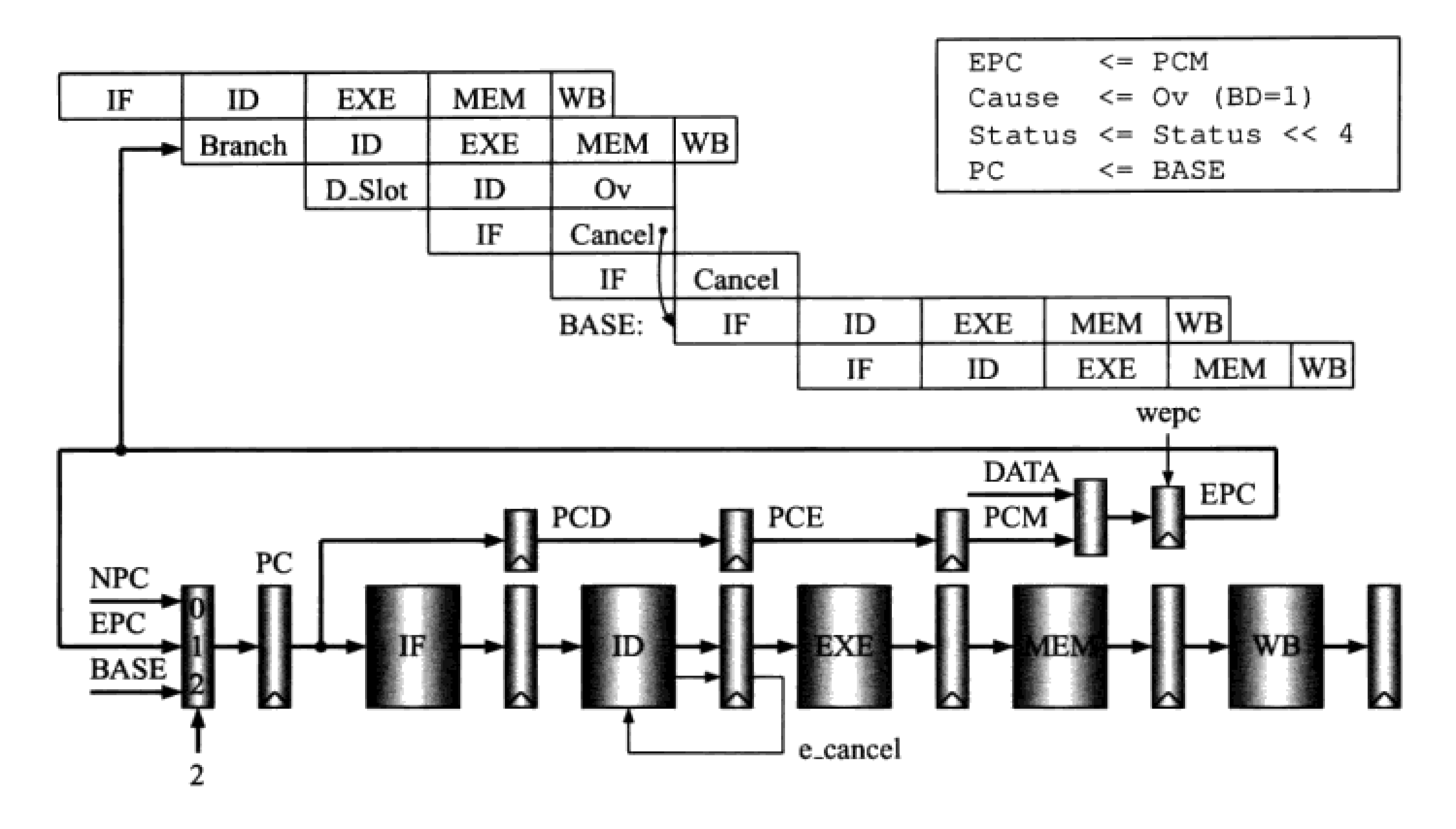


图 8.29 处在延迟槽的指令结果溢出时的流水线

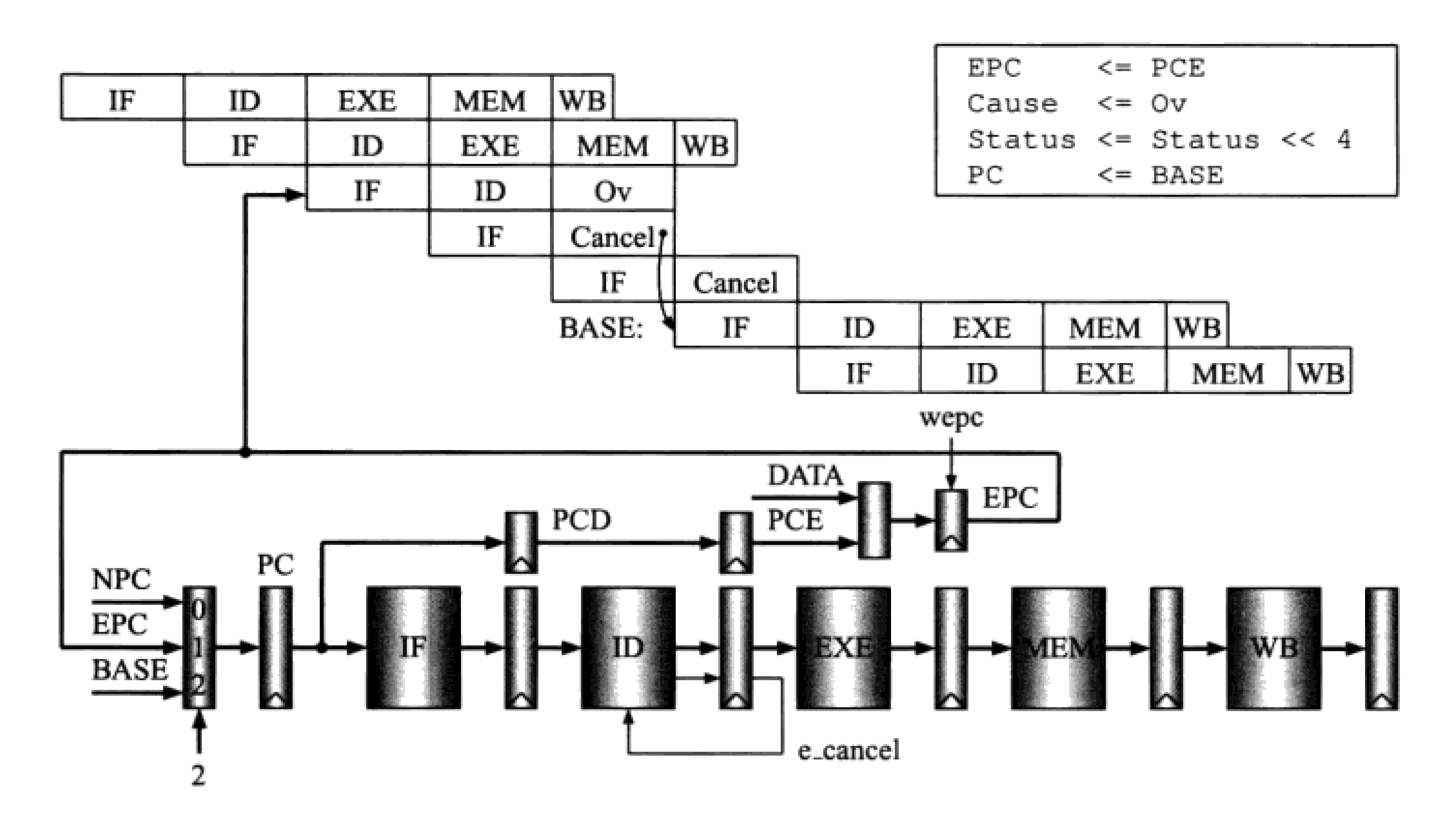


图 8.30 一般情况下的结果溢出时的流水线

表 8.2 异常和中断出现时写 EPC 各种情况的总结

助记符	ID 级是延迟转移指令	发生在延迟槽	其 他
Int	EPC ⇐ PCD	EPC ⇐ PC	EPC ⇐ PC
Sys	不可能	不允许	EPC ⇐ PCD
Unimpl	不可能	EPC ⇐ PCE	EPC ⇐ PCD
Ov	不出现	EPC ⇐ PCM	EPC ⇐ PCE

8.6 带有处理异常和中断功能的流水线 CPU 的设计

8.6.1 流水线 CPU 的总体结构

我们在图 8.17 的基础上实现精确中断和异常事件处理。图 8.31 示出新增电路部分，主要是围绕 Status、Cause 和 EPC 三个寄存器而设计的。注意这三个寄存器有各自的写使能端，它们分别是 wsta、wcau 和 wepc。

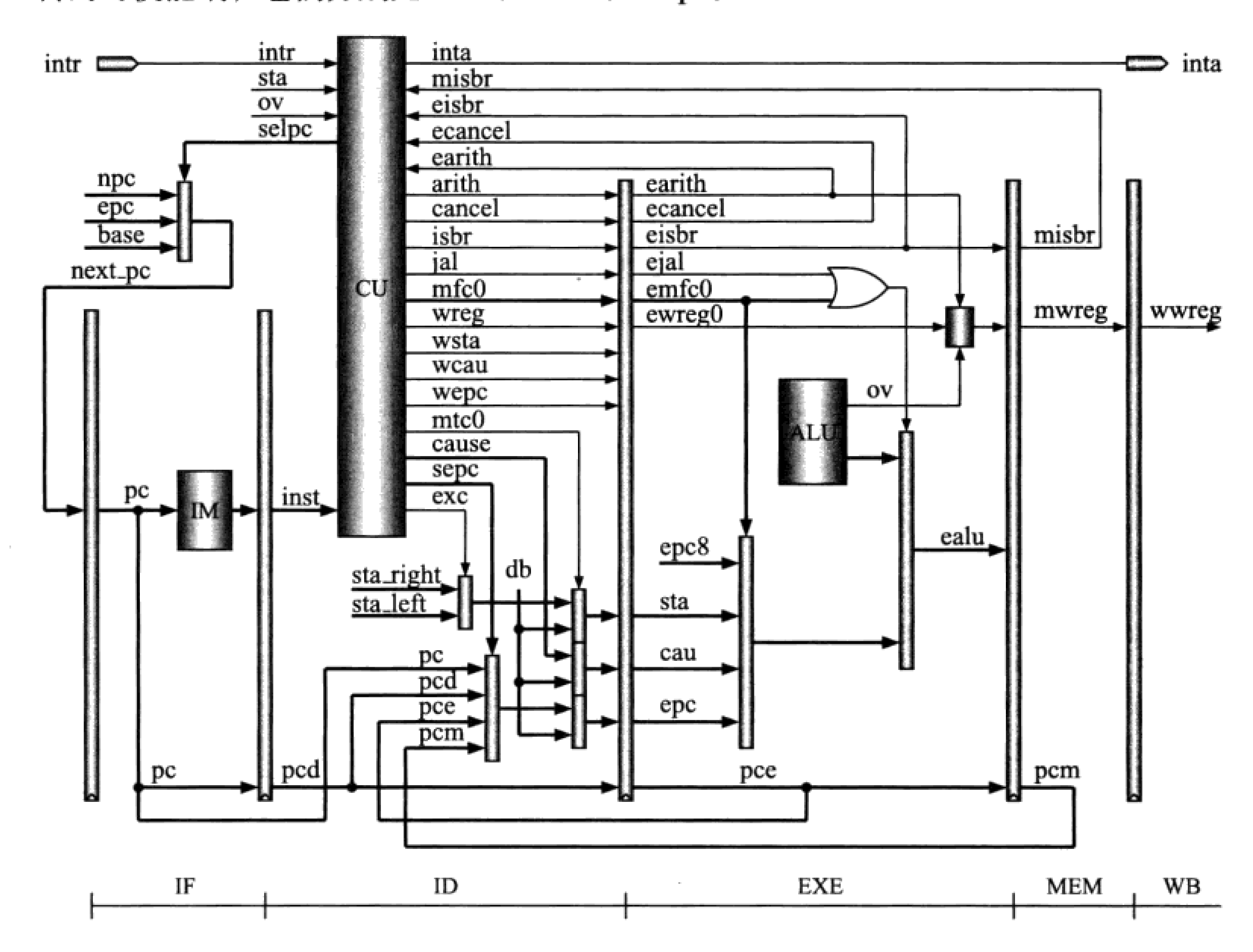


图 8.31 流水线 CPU 实现精确中断和异常事件处理的电路

通过前一节的讨论，我们已知在异常或中断发生时，可能写入 EPC 的数据有 PC、PCD、PCE 和 PCM。使用四选一多路器，我们可以得到多路器的选择信号 sepc[1:0]，如下。DATA (图 8.31 中的 db) 的选择用一个二选一多路器实现。

```
//              isbr         eisbr        misbr        others
// exc_int      PCD (01)     PC   (00)    PC   (00)    PC   (00)
// exc_sys      -            x            PCD (01)     PCD (01)
// exc_uni      -            PCE (10)     PCD (01)     PCD (01)
// exc_ovr      cancel       -            PCM (11)     PCE (10)

sepc[1] = exc_uni &  eisbr | exc_ovr;
```

```
sepc[0] = exc_int &   isbr | exc_sys |
          exc_uni & ~eisbr | exc_ovr & misbr;
```

CPU 可以使用 mfc0 和 mtc0 指令读写 Status、Cause 和 EPC 寄存器，其流水线见图 8.32。写 Status、Cause 和 EPC 在 ID 级完成，读 Status、Cause 和 EPC 与一般的 ALU 运算指令相同。

mfc0: | IF | ID | EXE | MEM | WB |　　mtc0: | IF | ID |

图 8.32 mfc0 和 mtc0 指令的流水线

8.6.2 流水线 CPU 的 Verilog HDL 代码

本小节给出能够精确处理异常和中断的流水线 CPU 的 Verilog HDL 代码。建议读者在阅读代码时要不时回过头来看看图 8.17 基本的流水线 CPU 的电路和图 8.31 有关处理异常和中断的电路。

```
module pipelined_cpu_exc_int (clock,memclock,resetn,pc,inst,ealu,malu,walu,
                              intr,inta);
    input clock,memclock,resetn,intr;
    output [31:0] pc,inst,ealu,malu,walu;
    output inta;

    parameter   EXC_BASE = 32'h00000008; // = base = BASE
    wire [31:0] bpc,jpc,npc,pc4,ins,pc4d,inst,da,db,imm,mmo,wdi;
    wire [4:0]  rn,ern;
    wire [3:0]  aluc;
    wire [1:0]  pcsource;
    wire        wpcir;
    wire        wreg,m2reg,wmem,aluimm,shift,jal;

 // PC
    dffe32 program_counter (next_pc,clock,resetn,wpcir,pc); // PC
    // IF
    cla32 pc_plus4 (pc,32'h4,1'b0,pc4); // PC+4
    mux4x32 nextpc (pc4,bpc,da,jpc,pcsource,npc); // Next PC
    lpm_rom  lpm_rom_component (.address(pc[7:2]),.q(ins));
    defparam lpm_rom_component.lpm_width    = 32,
             lpm_rom_component.lpm_widthad  = 6,
             lpm_rom_component.lpm_numwords = "unused",
             lpm_rom_component.lpm_file     = "sci_intr.mif",
             lpm_rom_component.lpm_indata   = "unused",
             lpm_rom_component.lpm_outdata  = "unregistered",
             lpm_rom_component.lpm_address_control = "unregistered";

 // pipeline registers: IF-ID
```

```
    dffe32 pc_4_r (pc4,clock,resetn,wpcir,pc4d); // PC+4 reg
    dffe32 inst_r (ins,clock,resetn,wpcir,inst); // IR
    dffe32 pcd_r  ( pc,clock,resetn,wpcir, pcd); // PCD reg
    wire [31:0] pcd;
    // ID
    wire [31:0] qa,qb;
    wire [1:0]  fwda,fwdb;
    wire [5:0]  op   = inst[31:26];
    wire [4:0]  rs   = inst[25:21];
    wire [4:0]  rt   = inst[20:16];
    wire [4:0]  rd   = inst[15:11];
    wire [5:0]  func = inst[5:0];
    assign      imm  = {{16{sext&inst[15]}},inst[15:0]};
    assign      jpc  = {pc4d[31:28],inst[25:0],2'b00};  // jump target
    regfile rf (rs,rt,wdi,wrn,wwreg,~clock,resetn,qa,qb); // reg file
    wire        regrt;
    mux2x5  des_reg_no (rd,rt,regrt,rn); // destination reg
    mux4x32 operand_a (qa,ealu,malu,mmo,fwda,da); // forward A
    mux4x32 operand_b (qb,ealu,malu,mmo,fwdb,db); // forward B
    wire  rsrtequ = ~|(da^db); // rsrtequ = (da == db)
    cla32 br_addr (pc4d,{imm[29:0],2'b00},1'b0,bpc); // branch target
    wire [1:0] sepc;
    wire       sext,ov,exc,mtc0,wepc,wcau,wsta,isbr,arith,cancel;
    cu_exc_int cu (mwreg,mrn,ern,ewreg,em2reg,mm2reg,rsrtequ,func,op,rs,rt,
                   rd,rs,wreg,m2reg,wmem,aluc,regrt,aluimm,fwda,fwdb,
                   wpcir,sext,pcsource,shift,jal,intr,sta,ecancel,ov,
                   earith,eisbr,misbr,inta,selpc,exc,sepc,cause,mtc0,wepc,
                   wcau,wsta,mfc0,isbr,arith,cancel);
    wire [31:0] sta,cau,epc,sta_in,cau_in,epc_in, // new for interrupt
                stalr,epcin,epc10,cause,pc8c0r,next_pc;
    wire [1:0]  mfc0,selpc;
    dffe32  c0_Status (sta_in,clock,resetn,wsta,sta); // Status register
    dffe32  c0_Cause  (cau_in,clock,resetn,wcau,cau); // Cause register
    dffe32  c0_EPC    (epc_in,clock,resetn,wepc,epc); // EPC register
    mux2x32 sta_mx (stalr,db,mtc0,sta_in); // mux for Status reg
    mux2x32 cau_mx (cause,db,mtc0,cau_in); // mux for Cause reg
    mux2x32 epc_mx (epcin,db,mtc0,epc_in); // mux for EPC reg
    mux2x32 sta_lr ({4'h0,sta[31:4]},{sta[27:0],4'h0},exc,stalr);
    mux4x32 epc_10 (pc,pcd,pce,pcm,sepc,epcin); // select epc source
    mux4x32 irq_pc (npc,epc,EXC_BASE,32'h0,selpc,next_pc); // for PC
    mux4x32 fromc0 (epc8,sta,cau,epc,emfc0,pc8c0r);  // for mfc0

// pipeline registers: ID-EXE
    reg [31:0]  ea,eb,eimm,epc4,pce;
    reg [4:0]   ern0;
    reg [3:0]   ealuc;
    reg         ewreg0,em2reg,ewmem,ealuimm,eshift,ejal;
```

```
    reg [1:0]    emfc0; // new
    reg          earith,ecancel,eisbr; // new
    always @(negedge resetn or posedge clock)
      if (resetn == 0) begin
          ewreg0  <= 0;           em2reg  <= 0;           ewmem   <= 0;
          ealuc   <= 0;           ealuimm <= 0;           ea      <= 0;
          eb      <= 0;           eimm    <= 0;           ern0    <= 0;
          eshift  <= 0;           ejal    <= 0;           epc4    <= 0;
          earith  <= 0;           ecancel <= 0;           eisbr   <= 0;
          emfc0   <= 0;           pce     <= 0;
      end else begin
          ewreg0  <= wreg;        em2reg  <= m2reg;       ewmem   <= wmem;
          ealuc   <= aluc;        ealuimm <= aluimm;      ea      <= da;
          eb      <= db;          eimm    <= imm;         ern0    <= rn;
          eshift  <= shift;       ejal    <= jal;         epc4    <= pc4d;
          earith  <= arith;       ecancel <= cancel;      eisbr   <= isbr;
          emfc0   <= mfc0;        pce     <= pcd;
      end
    // EXE
    wire [31:0] alua,alub,sa,ealu0,epc8;
    wire        zero;
    assign      sa = {eimm[5:0],eimm[31:6]};
    cla32   ret_addr (epc4,32'h4,1'b0,epc8);
    mux2x32 alu_ina  (ea,sa,eshift,alua);
    mux2x32 alu_inb  (eb,eimm,ealuimm,alub);
    mux2x32 save_pc8 (ealu0,pc8c0r,ejal|emfc0[1]|emfc0[0],ealu); // c0 regs
    assign  ern = ern0 | {5{ejal}};
    alu_ov  al_unit  (alua,alub,ealuc,ealu0,zero,ov);
    wire    ewreg = ewreg0 & ~(ov & earith); // cancel ov inst

// pipeline registers: EXE-MEM
    reg [31:0]    malu,mb,pcm; // new: pcm
    reg [4:0]     mrn;
    reg           mwreg,mm2reg,mwmem;
    reg           misbr; // new
    always @(negedge resetn or posedge clock)
        if (resetn == 0) begin
            mwreg   <= 0;           mm2reg  <= 0;           mwmem   <= 0;
            malu    <= 0;           mb      <= 0;           mrn     <= 0;
            misbr   <= 0;           pcm     <= 0;
        end else begin
            mwreg   <= ewreg;       mm2reg  <= em2reg;      mwmem   <= ewmem;
            malu    <= ealu;        mb      <= eb;          mrn     <= ern;
            misbr   <= eisbr;       pcm     <= pce;
        end
    // MEM
    lpm_ram_dq ram (.data(mb),.address(malu[6:2]),.we(mwmem & ~clock),
```

```
                .inclock(memclock),.outclock(memclock),.q(mmo));
    defparam   ram.lpm_width    = 32;
    defparam   ram.lpm_widthad = 5;
    defparam   ram.lpm_indata   = "registered";
    defparam   ram.lpm_outdata = "registered";
    defparam   ram.lpm_file     = "scd_intr.mif";
    defparam   ram.lpm_address_control = "registered";

 // pipeline registers: MEM-WB
    reg [31:0]     wmo,walu;
    reg [4:0]      wrn;
    reg            wwreg,wm2reg;
    always @(negedge resetn or posedge clock)
        if (resetn == 0) begin
            wwreg  <= 0;          wm2reg <= 0;          wmo    <= 0;
            walu   <= 0;          wrn    <= 0;
        end else begin
            wwreg  <= mwreg;      wm2reg <= mm2reg;     wmo    <= mmo;
            walu   <= malu;       wrn    <= mrn;
        end
    // WB
    mux2x32 wb_stage (walu,wmo,wm2reg,wdi);
endmodule
```

以下是控制单元的代码。

```
module cu_exc_int (mwreg,mrn,ern,ewreg,em2reg,mm2reg,rsrtequ,func,op,rs,rt,
                   rd,op1,wreg,m2reg,wmem,aluc,regrt,aluimm,fwda,fwdb,
                   wpcir,sext,pcsource,shift,jal,intr,sta,ecancel,ov,
                   earith,eisbr,misbr,inta,selpc,exc,sepc,cause,mtc0,wepc,
                   wcau,wsta,mfc0,isbr,arith,cancel);
    input  mwreg, ewreg, em2reg, mm2reg, rsrtequ;
    input [4:0] mrn, ern, rs, rt, rd, op1;
    input [5:0] func, op;
    output      wreg, m2reg,wmem, regrt, aluimm, sext, shift, jal;
    output [3:0] aluc;
    output [1:0] pcsource;
    output [1:0] fwda, fwdb; // forwarding
    output       wpcir;      // stall pipeline due to lw dependent

    // new for interrupt/exception
    input        intr,ecancel,ov,earith,eisbr,misbr;
    input [31:0] sta; // IM[3:0] : ov,unimpl,sys,int
    output       inta,exc,mtc0,wepc,wcau,wsta,isbr,arith,cancel;
    output [1:0] selpc,mfc0,sepc;
    output [31:0] cause;

    assign isbr     = i_beq | i_bne | i_j | i_jal;
```

```
assign arith     = i_add | i_sub | i_addi;
wire   overflow = ov & earith;
assign inta      = exc_int;
wire   exc_int   = sta[0] & intr;
wire   exc_sys   = sta[1] & i_syscall;
wire   exc_uni   = sta[2] & unimplemented_inst;
wire   exc_ovr   = sta[3] & overflow;
assign exc       = exc_int | exc_sys | exc_uni | exc_ovr;
assign cancel    = exc; // always cancel next inst
assign sepc[1]   = exc_uni &  eisbr | exc_ovr;
assign sepc[0]   = exc_int &   isbr | exc_sys |
                   exc_uni & ~eisbr | exc_ovr & misbr;
// ExcCode
// 0 0 : intr
// 0 1 : i_syscall
// 1 0 : unimplemented_inst
// 1 1 : overflow
wire   ExcCode0 = i_syscall | overflow;
wire   ExcCode1 = unimplemented_inst | overflow;
assign cause     = {eisbr,27'h0,ExcCode1,ExcCode0,2'b00}; // BD
assign mtc0  = i_mtc0;
assign wsta  = exc | mtc0 & rd_is_status | i_eret;
assign wcau  = exc | mtc0 & rd_is_cause;
assign wepc  = exc | mtc0 & rd_is_epc;
wire   rd_is_status = (rd == 5'd12); // cp0 Status register
wire   rd_is_cause  = (rd == 5'd13); // cp0 Cause register
wire   rd_is_epc    = (rd == 5'd14); // cp0 EPC register

// mfc0
// 0 0 : pc+8
// 0 1 : sta
// 1 0 : cau
// 1 1 : epc
assign mfc0[0] = i_mfc0 & rd_is_status | i_mfc0 & rd_is_epc;
assign mfc0[1] = i_mfc0 & rd_is_cause  | i_mfc0 & rd_is_epc;

// selpc
// 0 0 : npc
// 0 1 : epc
// 1 0 : EXC_BASE
// 1 1 : x
assign selpc[0] = i_eret;
assign selpc[1] = exc;

wire c0_type= ~op[5] & op[4] &~op[3] &~op[2] &~op[1] &~op[0];
wire i_mfc0 = c0_type &~op1[4] &~op1[3] &~op1[2] &~op1[1] &~op1[0];
wire i_mtc0 = c0_type &~op1[4] &~op1[3] & op1[2] &~op1[1] &~op1[0];
```

```
wire i_eret = c0_type & op1[4] &~op1[3] &~op1[2] &~op1[1] &~op1[0] &
        ~func[5] & func[4] & func[3] &~func[2] &~func[1] &~func[0];
wire i_syscall = r_type &~func[5] &~func[4] & func[3] & func[2] &
               ~func[1] &~func[0];
wire unimplemented_inst = ~(i_mfc0 | i_mtc0 | i_eret | i_syscall |
       i_add | i_sub | i_and | i_or | i_xor | i_sll | i_srl | i_sra |
       i_jr | i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_beq |
       i_bne| i_lui| i_j| i_jal);

wire r_type,i_add,i_sub,i_and,i_or,i_xor,i_sll,i_srl,i_sra,i_jr;
and(r_type,~op[5],~op[4],~op[3],~op[2],~op[1],~op[0]); // r format
and(i_add,r_type, func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_sub,r_type, func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_and,r_type, func[5],~func[4],~func[3], func[2],~func[1],~func[0]);
and(i_or, r_type, func[5],~func[4],~func[3], func[2],~func[1], func[0]);
and(i_xor,r_type, func[5],~func[4],~func[3], func[2], func[1],~func[0]);
and(i_sll,r_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_srl,r_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_sra,r_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_jr, r_type,~func[5],~func[4], func[3],~func[2],~func[1],~func[0]);
wire i_addi,i_andi,i_ori,i_xori,i_lw,i_sw,i_beq,i_bne,i_lui;
and(i_addi,~op[5],~op[4], op[3],~op[2],~op[1],~op[0]);
and(i_andi,~op[5],~op[4], op[3], op[2],~op[1],~op[0]);
and(i_ori, ~op[5],~op[4], op[3], op[2],~op[1], op[0]);
and(i_xori,~op[5],~op[4], op[3], op[2], op[1],~op[0]);
and(i_lw,   op[5],~op[4],~op[3],~op[2], op[1], op[0]);
and(i_sw,   op[5],~op[4], op[3],~op[2], op[1], op[0]);
and(i_beq, ~op[5],~op[4],~op[3], op[2],~op[1],~op[0]);
and(i_bne, ~op[5],~op[4],~op[3], op[2],~op[1], op[0]);
and(i_lui, ~op[5],~op[4], op[3], op[2], op[1], op[0]);
wire i_j,i_jal;
and(i_j,   ~op[5],~op[4],~op[3],~op[2], op[1],~op[0]);
and(i_jal, ~op[5],~op[4],~op[3],~op[2], op[1], op[0]);

wire i_rs = i_add  | i_sub | i_and  | i_or  | i_xor | i_jr  | i_addi |
            i_andi | i_ori | i_xori | i_lw  | i_sw  | i_beq | i_bne;
wire i_rt = i_add  | i_sub | i_and  | i_or  | i_xor | i_sll | i_srl  |
            i_sra  | i_sw  | i_beq  | i_bne | i_mtc0;
assign wpcir = ~(ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) |
                                                i_rt & (ern == rt)));
reg [1:0]    fwda, fwdb;
always @ (ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or rt)
    begin
        fwda = 2'b00; // default forward a: no hazards
        if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
            fwda = 2'b01; // select exe_alu
    end else begin
```

```
            if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
                fwda = 2'b10; // select mem_alu
            end else begin
                if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                    fwda = 2'b11; // select mem_lw
                end
            end
        end
        fwdb = 2'b00; // default forward b: no hazards
        if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
            fwdb = 2'b01; // select exe_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
                fwdb = 2'b10; // select mem_alu
            end else begin
                if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                    fwdb = 2'b11; // select mem_lw
                end
            end
        end
    end

    assign wmem    = i_sw & wpcir & ~ecancel & ~exc_ovr; //cancel next inst
    assign wreg    =(i_add | i_sub | i_and | i_or | i_xor | i_sll |
                     i_srl | i_sra | i_addi | i_andi | i_ori | i_xori |
                     i_lw | i_lui | i_jal | i_mfc0) &
                     wpcir & ~ecancel & ~exc_ovr; // cancel next inst
    assign regrt   = i_addi |i_andi |i_ori |i_xori |i_lw |i_lui |i_mfc0;
    assign jal     = i_jal;
    assign m2reg   = i_lw;
    assign shift   = i_sll  | i_srl  | i_sra;
    assign aluimm  = i_addi |i_andi |i_ori |i_xori |i_lw |i_lui |i_sw;
    assign sext    = i_addi | i_lw   | i_sw   | i_beq  | i_bne;
    assign aluc[3] = i_sra;
    assign aluc[2] = i_sub | i_or   | i_srl | i_sra | i_ori  | i_lui;
    assign aluc[1] = i_xor|i_sll|i_srl|i_sra|i_xori|i_beq|i_bne|i_lui;
    assign aluc[0] = i_and |i_or |i_sll |i_srl |i_sra |i_andi |i_ori;
    assign pcsource[1] = i_jr | i_j | i_jal;
    assign pcsource[0] = i_beq & rsrtequ | i_bne & ~rsrtequ | i_j | i_jal;
endmodule
```

8.6.3 异常和中断的测试程序与仿真波形

测试程序用第 6 章的 sci_intr.mif，测试数据用 scd_intr.mif。图 8.33 ~ 图 8.37 给出了中断出现的三种不同的情况，请检查每种情况下的返回地址。Sys、Unimpl 和 Ov 的波形就不再给出了。

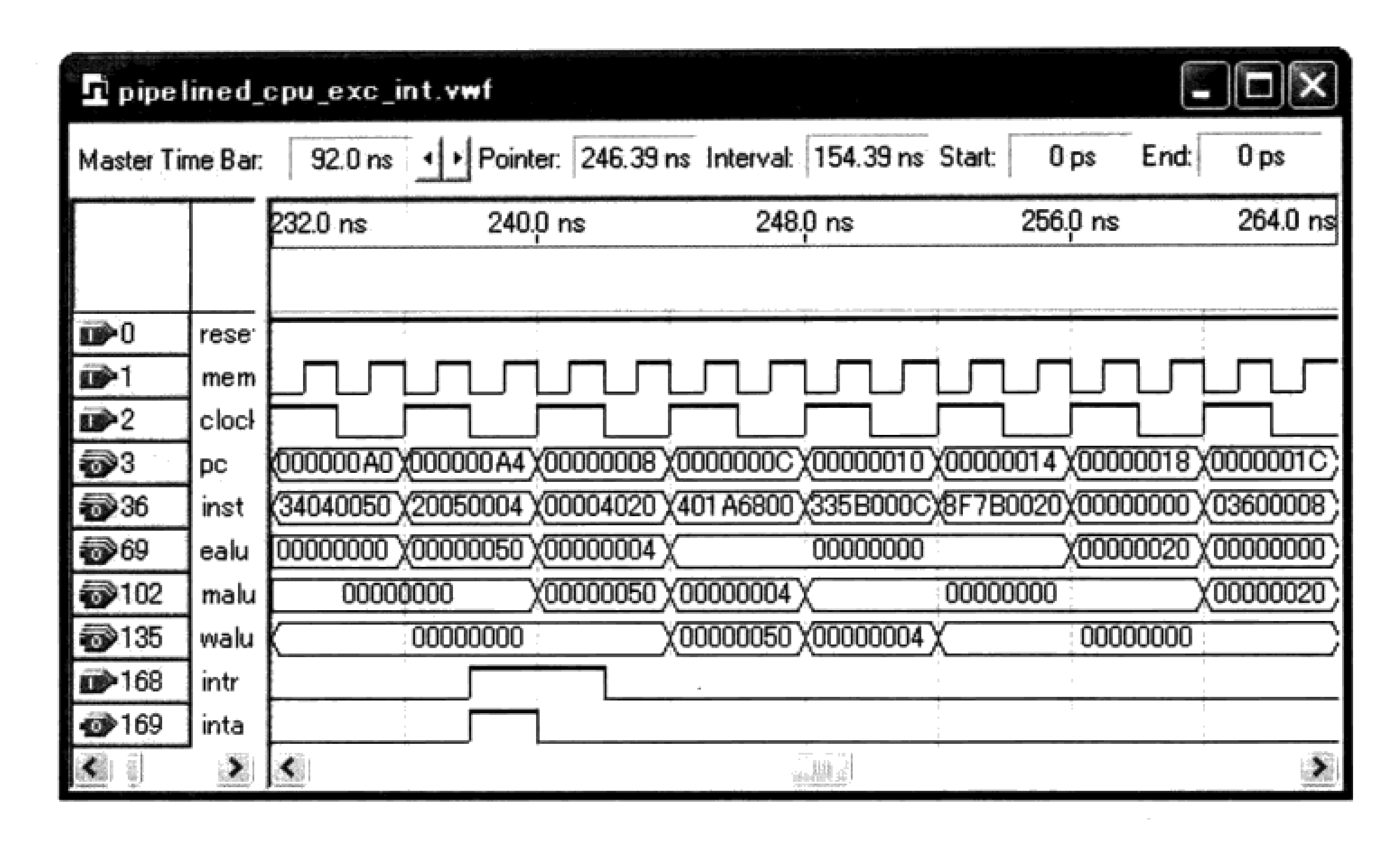


图 8.33 流水线 CPU 精确中断仿真波形图 (通常情况下的中断)

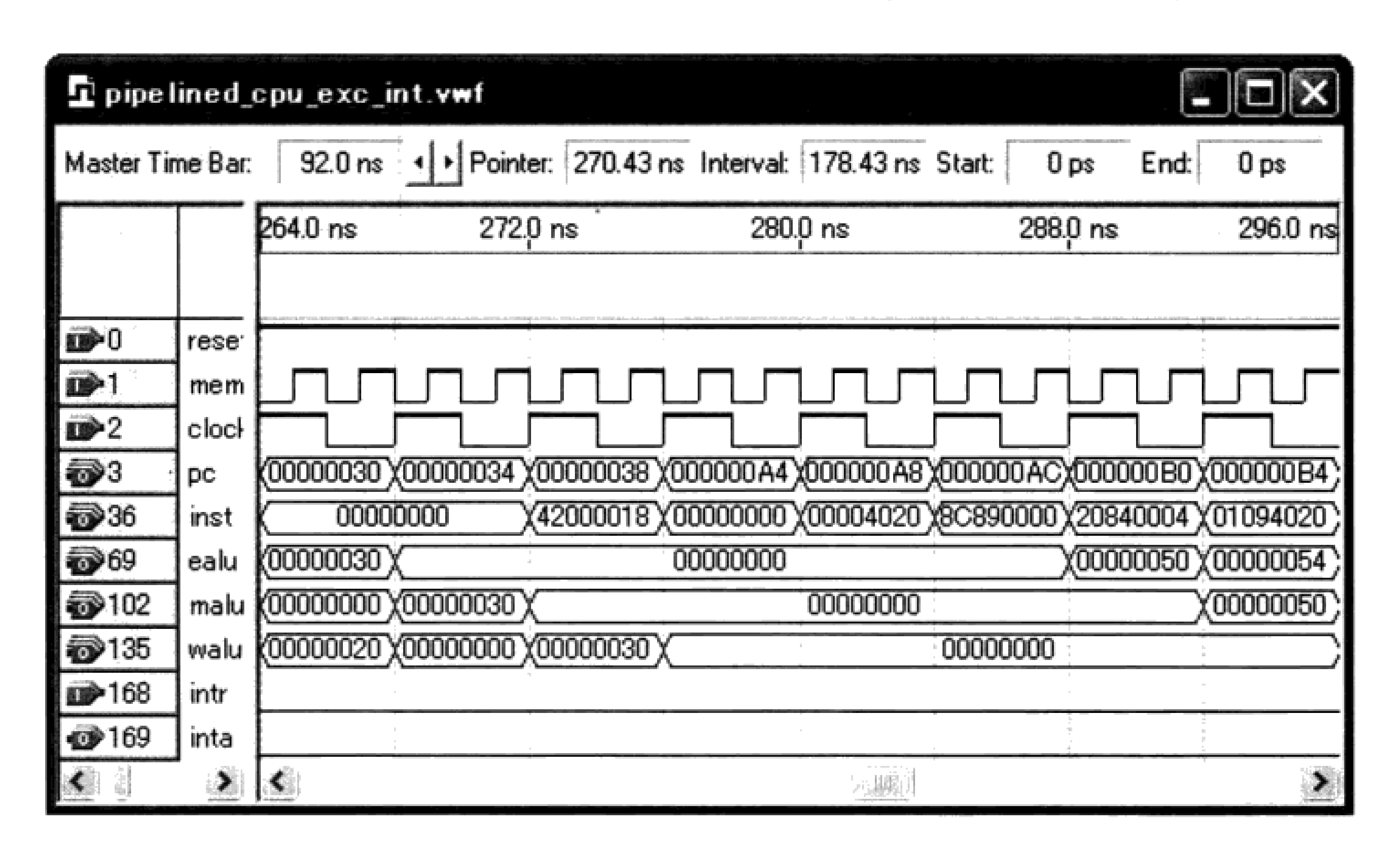


图 8.34 流水线 CPU 精确中断仿真波形图 (通常情况下的中断返回)

图 8.33 示出的是通常情况下的中断。中断发生时，ID 级的指令是 0x000000A0 处的 addi r5, r0, 4 指令。该指令被执行，而从 0x000000A4 处取来的指令被废弃。因此，执行完中断处理程序后，返回到 0x000000A4 (见图 8.34)。

图 8.35 示出的是在 ID 级执行 bne r5, r0, loop 指令时出现中断的情况。该指令的地址是 0x000000B8。这是一条转移指令，因此要被废弃。执行完中断处理程序后，

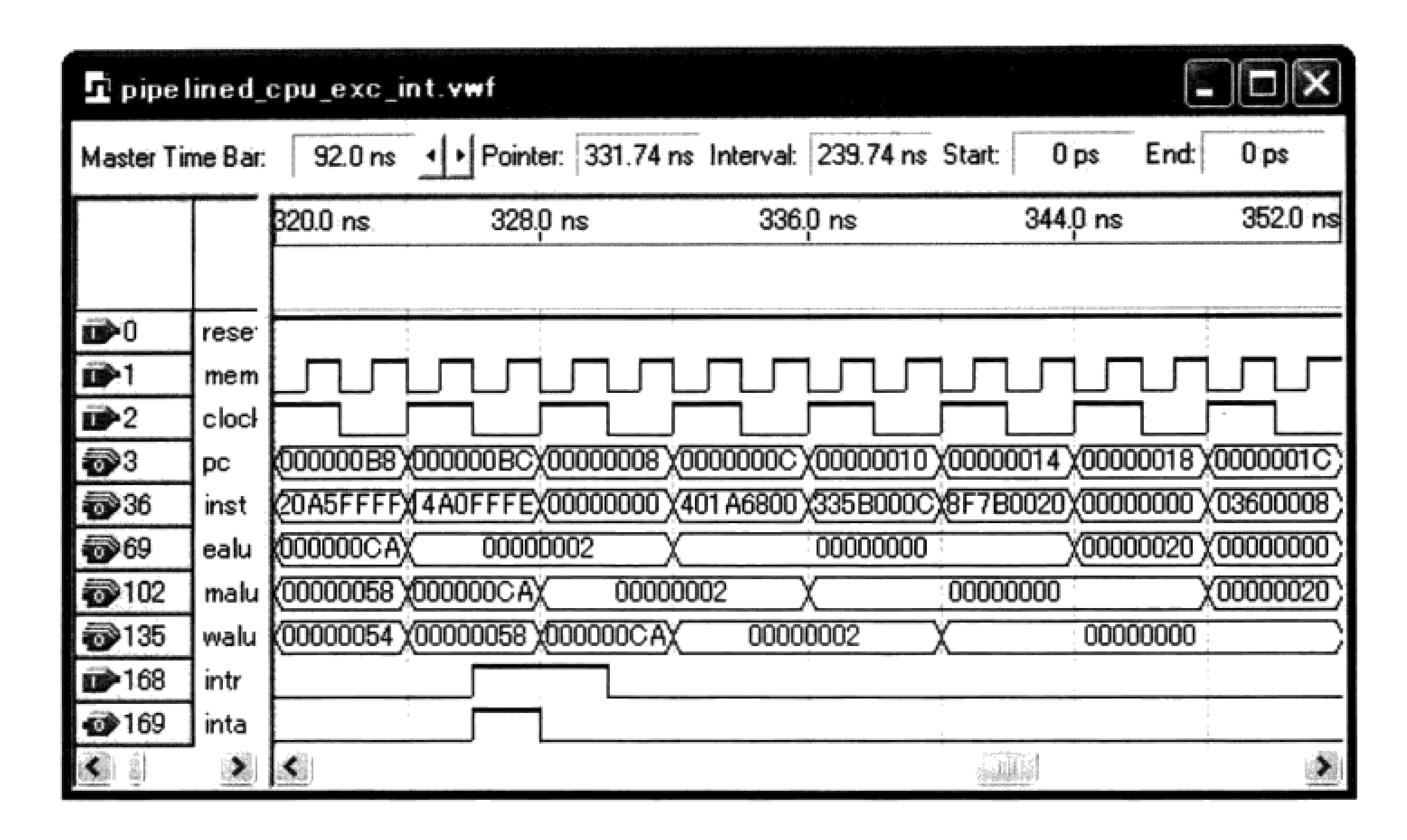


图 8.35　流水线 CPU 精确中断仿真波形图 (转移指令处中断)

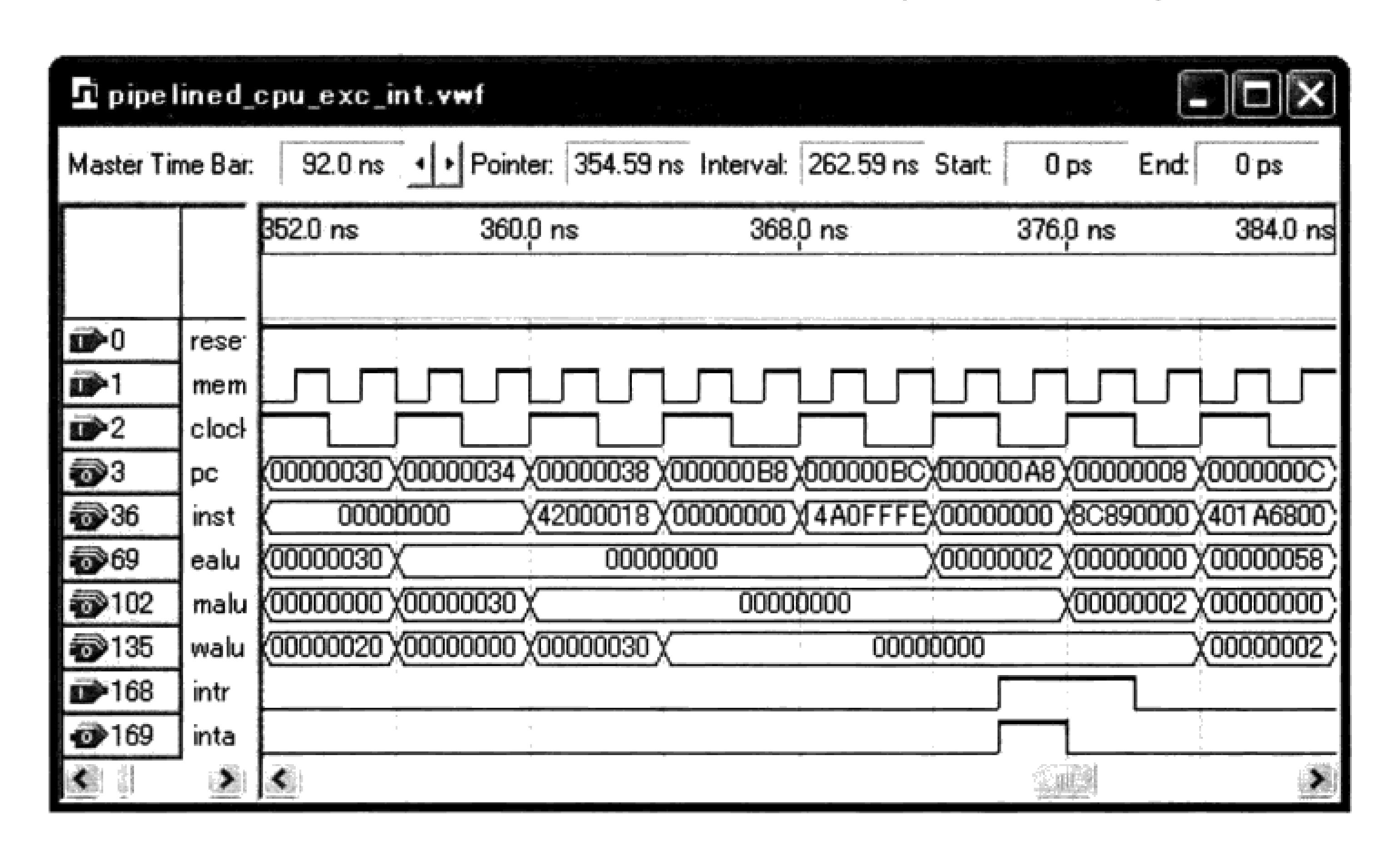


图 8.36　流水线 CPU 精确中断仿真波形图 (转移中断返回及延迟槽处中断)

要返回到 0x000000B8 处，重新执行 bne r5, r0, loop 指令。

图 8.36 示出了在 ID 级执行延迟槽中的指令时出现中断的情况。转移指令是 0x000000B8 处的 bne r5, r0, loop 指令。因为我们的方案是把转移指令 bne 和其延迟槽中的指令 nop 都执行完，因此执行完中断处理程序后，要返回到 0x000000A8 处，即 bne 的转移目标地址 loop，见图 8.37。

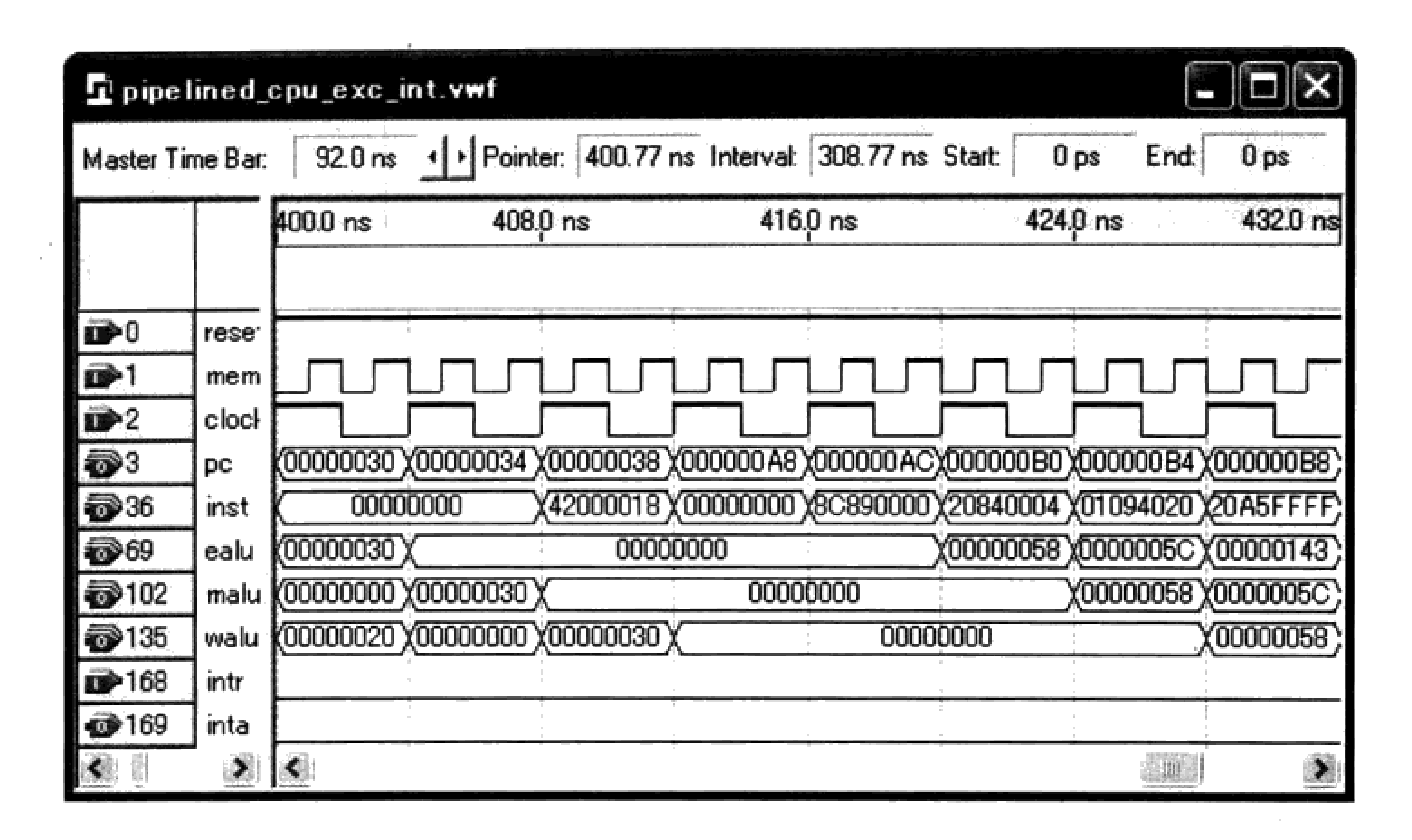


图 8.37 流水线 CPU 精确中断仿真波形图 (延迟槽中断返回)

8.7 习题

1. 设一个流水线 CPU 的流水线级数为 m，每一级占用一个时钟周期。(1) 计算该 CPU 执行 n 条指令所需的周期数；(2) 给出 CPI 的公式；(3) 当 n 比 m 大很多时，求 CPI。
2. 为什么在流水线 CPU 中会出现数据相关和转移相关的问题？简述解决这些问题的思路和方法。
3. 在图 8.19 的流水线 CPU 仿真波形的 56ns ~ 60ns 期间，ealu 的值为什么是 0x00000146?
4. 用逻辑图输入的方法设计本章的流水线 CPU。
5. 本章给出的流水线 CPU 的 Verilog HDL 代码总体来讲具有结构描述风格。试从顶层开始就使用功能描述风格的 Verilog HDL 重新设计流水线 CPU。
6. 重新设计异常处理电路：允许 syscall 指令出现在延迟槽中。
7. 修改 sci_intr.mif，使其能够测试 Unimpl 和 Ov 发生在延迟槽时的情况。
8. 使用一个带有置 1 和清零端的 D 触发器来保存外部中断请求，重新设计中断响应电路，使其只在一般情况下响应中断。即在 ID 级的指令是转移或跳转指令或 ID 级处在延迟槽的情况下不响应中断。注意，当外部电路收到中断确认后，应主动撤销中断请求。
9. 实现一个计时器中断。
10. 调查其他的实现精确中断的方法并写出一份报告。
11. 调查转移预测 (Branch Prediction) 技术并写出一份报告。

第 9 章　浮点算法及 FPU Verilog HDL 设计

到目前为止，我们设计的 CPU 中只有整数部件 IU (Integer Unit)。如果要对小数进行计算，则需要有相应的电路支持。支持小数计算的电路有两种：定点部件和浮点部件。定点部件与整数部件没有本质的区别，只是要假设有一个小数点存在于某两位数之间，计算时要把小数点对齐。

浮点部件 FPU (Floating Point Unit) 与定点部件不同，因为浮点数的小数点不是固定的 (浮点的全称是浮动小数点)。当然，浮点数也是用二进制数来表示的，但这些二进制数位有不同的意义。以前各有名的公司，如 IBM 等，都定义了自己的浮点数格式。由于各公司的浮点数格式不同，使得程序和数据不能共享。自 IEEE 754 浮点标准宣布之后，各公司基本上都放弃了自己的格式，而采用了 IEEE 754 标准。

本章首先简要介绍 IEEE 754 浮点格式的定义，然后给出单精度浮点数与整数之间的转换算法以及单精度浮点数的加、减、乘、除和开方的算法。所有的算法均给出 Verilog HDL 的实现源代码。

9.1　IEEE 754 浮点数格式

IEEE 754 标准[5] 主要定义了单精度和双精度两种浮点数格式。单精度浮点数使用 32 位二进制数来表示，而双精度浮点数用 64 位。如果我们在 Java 或 C 语言程序中分别用 float 和 double 定义两个变量，例如：

```
float s = -1.75;
double x = 1.75;
```

则编译器会把它们分别转换成单精度和双精度浮点数。以下我们重点介绍单精度浮点数格式。图 9.1 给出了 32 位 IEEE 754 单精度浮点数格式。它由 3 部分组成：第 31 位是符号位 s (Sign)，0 代表正数，1 代表负数；第 30 ～ 23 位是 8 位阶码位 e (Exponent)；第 22 ～ 0 位是 23 位尾数位 f (Fraction)。



图 9.1　IEEE 754 单精度浮点数格式

如果 $0 < e < 255$，单精度浮点数的值为 $(-1)^s \times 2^{e-127} \times 1.f$。我们称其为规格化数。注意在计算规格化数值的公式中有一项为 1.f，其中小数点前面的 1 称为隐藏位，它并不占用 32 位中的任何一位。阶码用移码表示，偏移值为 127。例如，假设我们有一个单精度浮点数，它的 32 位二进制表示是

1_01111111_11000000000000000000000

则该浮点数的值为 $(-1)^1 \times 2^{127-127} \times 1.11 = -(1+0.5+0.25) = -1.75$。也就是说，编译器会把 `float s = -1.75` 转换成上面的 32 位二进制数。

绝对值最小的规格化数为 x_00000001_00000000000000000000000，它的绝对值是 $2^{-126} \times 1.0$。如果要表示一个比它还要小的非 0 数值，我们可以用所谓非规格化格式：如果 e = 0，单精度非规格化浮点数的值为 $(-1)^s \times 2^{-126} \times 0.f$。例如，1_00000000_11000000000000000000000 的数值为 $-2^{-126} \times 0.75$。

如果 e = 0 并且 f = 0，它的数值为 +0 或 −0。如果 e = 255 并且 f = 0，它表示 $+\infty$ (无穷大) 或 $-\infty$。如果 e = 255 并且 $f \neq 0$，它表示它自己不是一个数 (NaN，Not a Number)。NaN 是需要的，例如 $\infty - \infty = \text{NaN}$。

IEEE 754 双精度浮点数格式用 64 位二进制数表示，它的阶码占用 11 位，尾数有 52 位。规格化数值为 $(-1)^s \times 2^{e-1023} \times 1.f$，非规格化数值为 $(-1)^s \times 2^{-1022} \times 0.f$。其余与单精度浮点数的定义类似。

9.2 单精度浮点数与整数之间的转换

浮点数与整数之间的转换是我们在编写程序时经常用到的操作，本节讨论单精度浮点数与用 32 位补码表示的整数之间的转换算法并给出 Verilog HDL 源代码。

9.2.1 浮点数转换成整数

我们知道，32 位补码表示的整数 d 满足 $-2^{31} \leqslant d \leqslant +2^{31}-1$，而单精度浮点数所能表示的数的范围比 32 位整数大得多，因此很多浮点数是不能被转换成整数的。以下我们通过一个例子说明转换方法。

设浮点数 $a = 4effffff_{16} = 0_10011101_11111111111111111111111_2$，则 $s_a = 0$，$e_a = 127 + 30$，$1.f_a = 1.11111111111111111111111_2$。由 e_a 的值可知，我们要把 $1.f_a$ 的小数点右移 30 位。因此整数结果 $d = 01111111111111111111111110000000_2 = 7fffff80_{16} = 2147483520_{10}$。这也是能够正确转换成整数的最大浮点数。

由上例我们总结出单精度浮点数转换成 32 位整数的方法：首先在 24 位的 $1f_a$ 的右边补 8 个 0，使它变成 32 位整数。即我们已经把小数点右移了 31 位。如果 $e_a < 127 + 31$，我们还要把这个 32 位整数右移。因此我们确定必须右移的位数为 $127 + 31 - e_a$。如果浮点数为负，我们还要把它转换成负数 (取反加 1)。

如果浮点数超出了 $-2^{31} \sim (+2^{31}-1)$ 的范围，我们令 $d = 80000000_{16}$ 并把标志 invalid 置 1，表示结果无效。另外，由于浮点数能表示小数，如 a = 0.5，而整数不能，因此我们还给出标志位 precision_lost，表示精度已经损失。以下给出一个精度损失的例子。设 $a = 3fc00000_{16} = 0_01111111_10000000000000000000000_2$，则 $s_a = 0$，$e_a = 127$，$1f_a = 110000000000000000000000_2$。由于要把 $1f_a$ 右移 $127 + 31 - e_a = 31$ 位，结果变为 $0000000000000000000000000000001.1_2$。小数点右面的 1 将被遗弃，造成精度损失。表 9.1 给出了 12 个单精度浮点数转换成 32 位整数的例子，其中的输出信号 denormalized 表示浮点数为非规格化数。

表 9.1 单精度浮点数转换成 32 位整数举例

单精度浮点数	4effffff	4f000000	3f800000	3f000000	00000001	3fc00000
32 位整数	7fffff80	80000000	00000001	00000000	00000001	00000001
precision_lost	0	0	0	1	1	1
denormalized	0	0	0	0	1	0
invalid	0	1	0	0	0	0
单精度浮点数	cf000000	cf000001	bf800000	bf7fffff	80000001	00000000
32 位整数	80000000	80000000	ffffffff	00000000	00000000	00000000
precision_lost	0	0	0	1	1	0
denormalized	0	0	0	0	1	0
invalid	0	1	0	0	0	0

以下是 Verilog HDL 源代码。为了能够检测出精度是否损失，我们在移位过程中使用了 56 位数据格式，其中左 32 位是结果，右 24 位是被移出的位。若右 24 位不是全 0，则表示结果精度损失。

```
module f2i (a,d,precision_lost,denormalized,invalid);
    input  [31:0] a;        // float
    output [31:0] d;        // int range: -2^{31} ~ +2^{31}-1
    output precision_lost;  // => 00000000
    output denormalized;    // => 00000000
    output invalid;         // (inf,nan,out_of_range) --> 80000000
    reg [31:0] d;
    reg precision_lost;
    reg invalid;
    wire hidden_bit = |a[30:23];
    wire frac_is_not_0 = |a[22:0];
    assign denormalized = ~hidden_bit & frac_is_not_0;
    wire is_zero = ~hidden_bit & ~frac_is_not_0;
    wire sign = a[31]; // sign
    wire [8:0] shift_right_bits = 9'b010011110 - {1'b0,a[30:23]};
    wire [55:0] frac0 = {hidden_bit,a[22:0],32'h0};
    wire [55:0] f_abs = ($signed(shift_right_bits) > 9'h20)?
                        frac0 >> 6'h20 : frac0 >> shift_right_bits;
    wire lost_bits = |f_abs[23:0];
    wire [31:0] int32 = (sign)? -f_abs[55:24] : f_abs[55:24];
    always @ * begin
        if (denormalized) begin //den
            precision_lost = 1;
            invalid = 0;
            d = 32'h00000000;
        end else begin // not den
            if (shift_right_bits[8]) begin // too big
                precision_lost = 0;
```

```
                invalid = 1;
                d = 32'h80000000;
            end else begin // shift right
                if (shift_right_bits[7:0] > 8'h1f) begin//too small
                    if (is_zero) precision_lost = 0;
                    else precision_lost = 1;
                    invalid = 0;
                    d = 32'h00000000;
                end else begin
                    if (sign != int32[31]) begin // out of range
                        precision_lost = 0;
                        invalid = 1;
                        d = 32'h80000000;
                    end else begin // normal case
                        if (lost_bits) precision_lost = 1;
                        else precision_lost = 0;
                        invalid = 0;
                        d = int32;
                    end
                end
            end
        end
    end
endmodule
```

单精度浮点数转换成 32 位整数的 Verilog HDL 代码的仿真结果如图 9.2 所示。

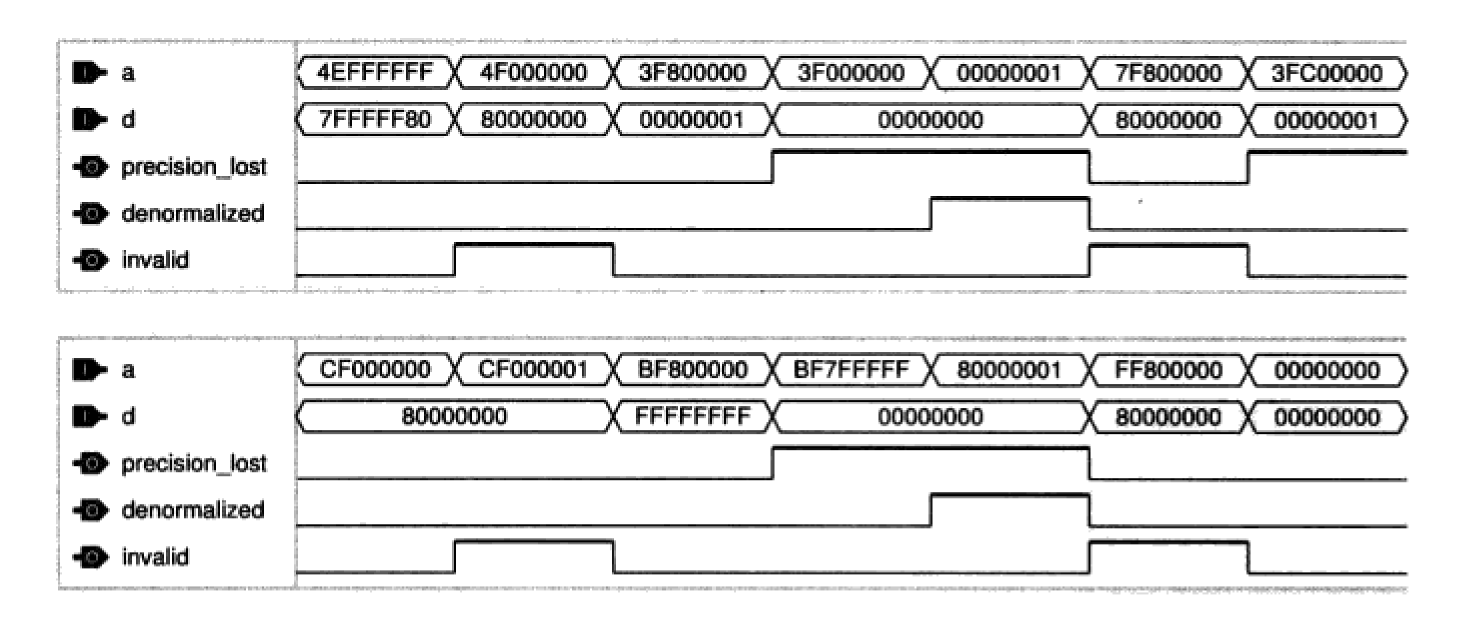


图 9.2 单精度浮点数转换成 32 位整数仿真结果

为了验证以上代码的正确性，作者用 C 写了一个程序，用来测试 x86 FPU 浮点数到整数的转换结果。该程序首先从键盘读入一个十六进制表示的 32 位浮点数，然后使用 x86 汇编指令 fldcw 把一个 16 位的浮点控制字写入浮点控制寄存器。浮点控制寄存器每位的意义在图 9.3 中给出，主要完成的任务是屏蔽 FPU 产生的异常。

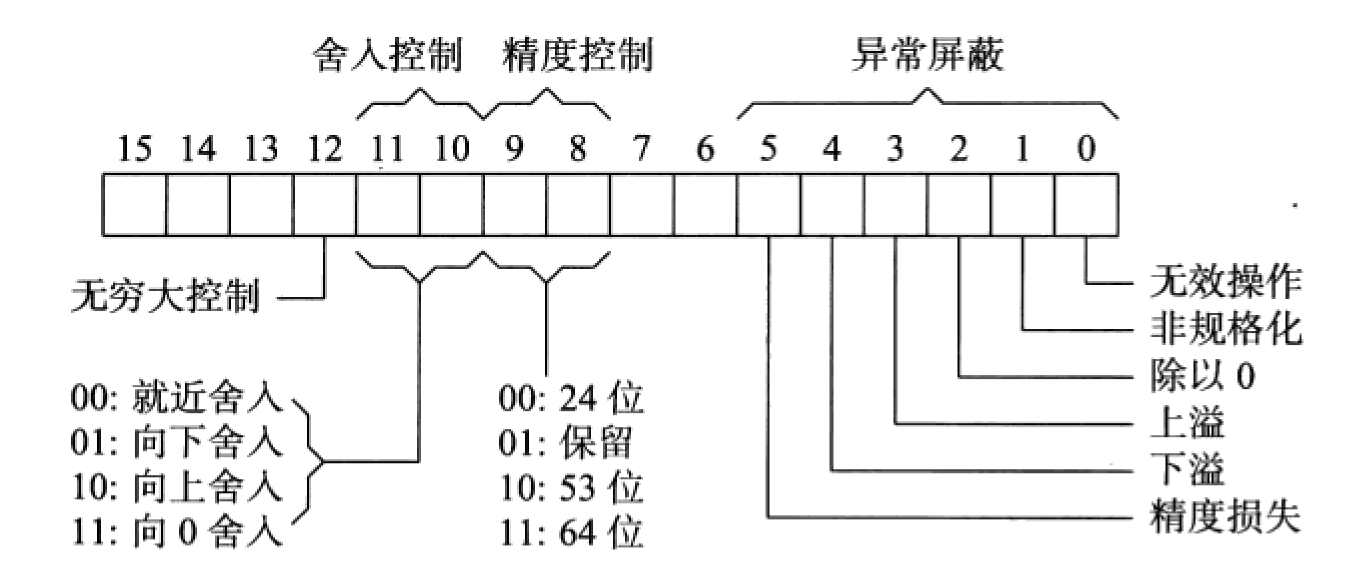


图 9.3 FPU 浮点控制寄存器 (x86)

在完成浮点数到整数的转换之后，使用 fstsw 汇编指令读取状态寄存器，以检查有哪些异常出现。浮点状态寄存器的格式在图 9.4 中简要给出。在转换之前，我们使用了 fclex 指令来清除状态寄存器中的所有异常标志。

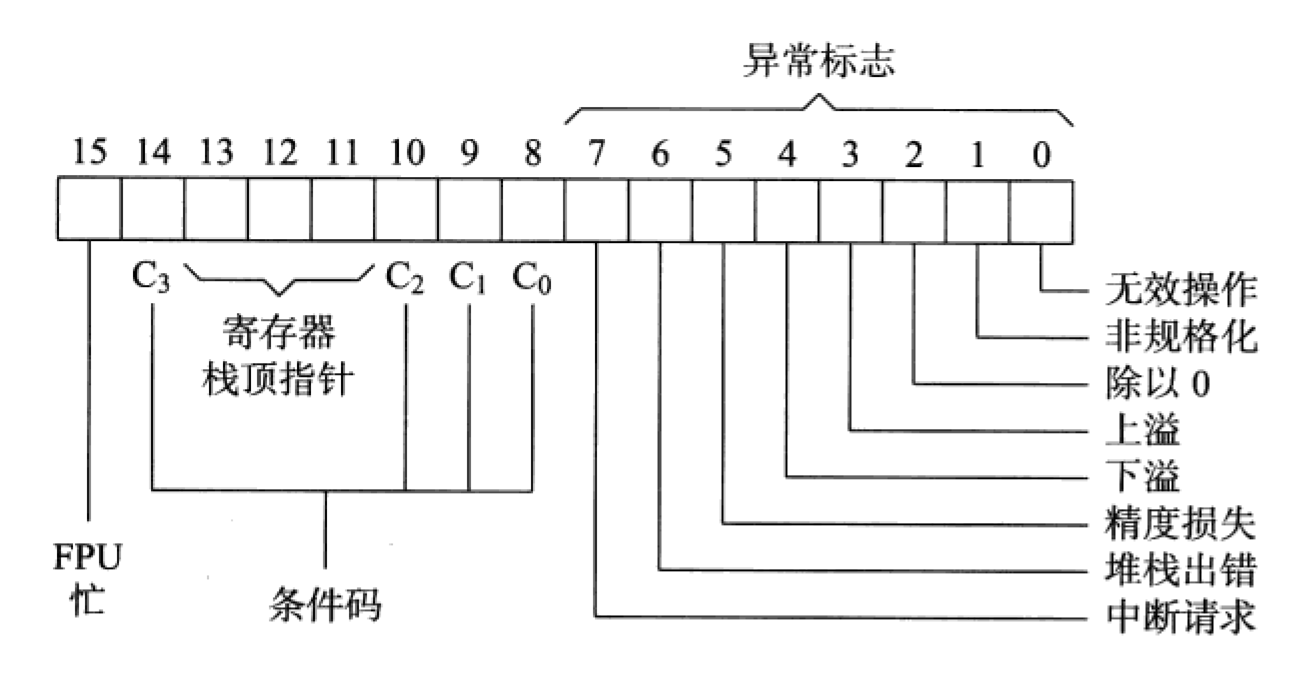


图 9.4 FPU 浮点状态寄存器 (x86)

测试 x86 FPU 浮点数到整数的转换结果的 C 语言程序如下。我们使用了 union 结构来实现对同一数据以不同方式的访问 (以十六进制格式输入浮点数)。

```
// f2i test for (x86 + Linux + gcc)
#include<stdio.h>
#define Chop_disable_exception \
asm volatile("fldcw _RoundChop_disable_exception")  // 写 FPU 控制字
int _RoundChop_disable_exception = 0x1c3f;          // 屏蔽异常
#define Read_fp_state_word \
asm volatile("fstsw _fp_state_word")                // 读 FPU 状态字
int _fp_state_word;                                 // FPU 状态结果
#define Clear_exceptions_of_sw \
asm volatile("fclex")                               // 清除状态寄存器
```

```
int main(void) {
    union {
        int intword;
        float floatword;
    } u;
    int d;
    while (1) {
        fprintf (stderr,"input a float number in hex format: ");
        fscanf (stdin,"%x",&u.intword); // 读一个浮点数进来
        Chop_disable_exception;         // 写FPU控制字: 屏蔽异常
        Clear_exceptions_of_sw;         // 清除状态寄存器所有异常位
        d = u.floatword;                // 浮点数转换成整数
        fprintf (stderr,"f = %08X = %0.6f\n",
                 u.intword, u.floatword);
        fprintf (stderr,"d = %08X = %08d\n", d, d); // 显示整数
        Read_fp_state_word;             // 读FPU状态字并显示状态结果
        fprintf (stderr,"fp_state_word = %04X\n",_fp_state_word);
    }
}
```

我们可以用 gcc 对上述程序进行编译。程序的执行结果的例子如下所示。第一个输入数据是十进制的 1.5，转换成整数后精度损失，所以状态寄存器的第 5 位设置为 1。第二个数据是能转换的、不产生任何异常的最大正数。

```
[yamin@localhost cpu]$ gcc f2i.c -o f2i
[yamin@localhost cpu]$ f2i
input a float number in hex format: 3fc00000
f = 3FC00000 = 1.500000
d = 00000001 = 00000001
fp_state_word = 0020
input a float number in hex format: 4effffff
f = 4EFFFFFF = 2147483520.000000
d = 7FFFFF80 = 2147483520
fp_state_word = 0000
input a float number in hex format:
[yamin@localhost cpu]$
```

9.2.2 整数转换成浮点数

所有的整数都能转换成浮点数，不会出现“无效操作”的情况。但是，由于整数有 32 位，而单精度浮点数的尾数只有 24 位 (算上隐藏位)，因此会出现精度损失的情况。我们以 $d = 1fffffff_{16} = 00011111111111111111111111111111_2$ 为例，说明整数到浮点数的转换方法。转换后的结果为 IEEE 754 格式的单精度浮点数，用 a 表示。

由于 d 是整数，可以假设有一个小数点在第 0 位 (最右位) 的右边。现在我们把小数点左移 31 位，即移到第 31 位与第 30 位之间。这相当于把 d 右移了 31 位。为

了保持 d 的数值大小不变，我们应设它的阶码 31，即 $d = (d \times 2^{-31}) \times 2^{31}$，其中括号部分是把小数点左移 31 位。然后我们再把该数值规格化，即把它左移，使其最高位为 1。我们用 shift_amount 表示移位位数，用 f0 表示移位后的数据。该例中 shift_amount = 3，$f0 = 11111111111111111111111111111000_2$。

为了保持数值大小不变，左移的位数 shift_amount 要从阶码中减掉。因此，IEEE 754 格式中 a 的阶码 $e_a = 31 - \text{shift_amount} + 127$。即 $e_a = 158 - \text{shift_amount}$。小数点左边的 1 (最高位) 即为隐藏位，从小数点开始往右数出 23 位即得到尾数 f_a。我们这里简单地把剩下的最右 8 位扔掉。如果被扔掉的 8 位不是全 0，我们设输出信号 precision_lost = 1，意味着精度受损了。以下是该算法的 Verilog HDL 代码，注意 shift_amount 的产生方法。

```
module i2f (d,a,precision_lost);
    input  [31:0] d;  // int range: -2^{31} ~ +2^{31}-1
    output [31:0] a;  // float
    output precision_lost;
    wire sign = d[31]; // sign
    wire [31:0] f5 = sign? -d : d;
    wire [31:0] f4,f3,f2,f1,f0;
    wire [4:0] shift_amount;

    assign shift_amount[4] = ~|f5[31:16];                    // 16-bit 0
    assign f4 = shift_amount[4]? {f5[15:0],16'b0} : f5;
    assign shift_amount[3] = ~|f4[31:24];                    //  8-bit 0
    assign f3 = shift_amount[3]? {f4[23:0], 8'b0} : f4;
    assign shift_amount[2] = ~|f3[31:28];                    //  4-bit 0
    assign f2 = shift_amount[2]? {f3[27:0], 4'b0} : f3;
    assign shift_amount[1] = ~|f2[31:30];                    //  2-bit 0
    assign f1 = shift_amount[1]? {f2[29:0], 2'b0} : f2;
    assign shift_amount[0] = ~f1[31];                        //  1-bit 0
    assign f0 = shift_amount[0]? {f1[30:0], 1'b0} : f1;
    wire [22:0] fraction = f0[30:8];
    assign precision_lost = |f0[7:0];
    wire [7:0] exponent = 8'b1001_1110 - {3'h0,shift_amount};
    assign a = (d == 0)? 0 : {sign,exponent,fraction};
endmodule
```

32 位整数转换成单精度浮点数的 Verilog HDL 代码的仿真结果如图 9.5 所示。

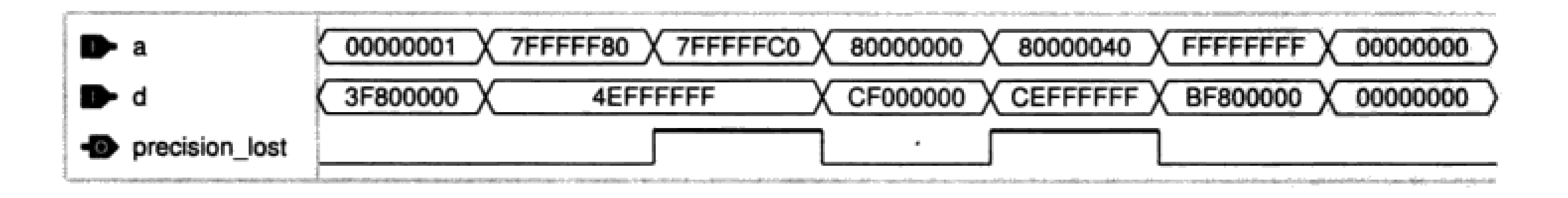


图 9.5　32 位整数转换成单精度浮点数仿真结果

9.3 浮点加法器 FADD 设计

虽然叫浮点加法器，它其实也能做减法。本节首先讨论浮点加法算法，然后给出 Verilog HDL 源代码，最后描述流水线浮点加法器的设计方法。

9.3.1 浮点加法算法

我们首先通过一个具体的例子来讲述浮点加法算法。假设有以下两个 IEEE 754 单精度浮点数 A 和 B，试计算 $C = A + B$。

```
A: 0 01111000 11000000000000000010001 (十六进制：3C600011)
B: 1 01111101 00000100000000000000000 (十六进制：BE820000)
```

这是两个规格化数，它们的隐藏位均为 1。虽然我们要计算 $A + B$，但从两个数的符号位看出，A 是正数而 B 是负数。因此我们要把它们的绝对值相减。谁减谁？当然是大的绝对值减去小的绝对值。哪个绝对值大？比较一下它们的阶码就知道 B 的绝对值大。注意这时还不能对尾数直接相减，因为两个数的阶码不同，它们的差值为 $01111101_2 - 01111000_2 = 00000101_2 = 5_{10}$。

首先我们把绝对值小的阶码调整成与绝对值大的阶码相同的值，即把 A 的阶码加上 5。为了保证 A 的绝对值大小不变，我们必须把 A 的尾数连同隐藏位一起向右移 5 位。为了保证计算精度，IEEE 754 标准规定在计算时要在尾数的右端留出 3 位附加位。这 3 位附加位分别有自己的名字，从左至右为 G (Guard)，R (Round) 和 S (Sticky)。我们用 GRS 称呼它们。绝对值大的数的 GRS 位为 000。当我们右移绝对值小的尾数时，如果 R 位右边的所有位中有任何一位为 1 时，设置 S 为 1。

另外，考虑到尾数的运算结果可能是原来一个尾数的两倍，在最高位处还要再加 1 位。因此总的运算位数等于 $1 + 1 + 23 + 3 = 28$。以下是尾数的计算：

```
                                  GRS
    01.00000100000000000000000    000   (B 的尾数)
 -  00.00000111000000000000000    101   (A 的尾数)
 ----------------------------------------------------
    00.11110101111111111111111    011
```

为了得到用 IEEE 754 单精度浮点数格式表示的结果，我们还必须把尾数相减的结果 (`00.11110101111111111111111 011`) 转换成 1.f 的格式。将它左移一位，变成 (`01.11101011111111111111110 110`)，同时要从阶码中减去 1，变成 01111100，以保证结果大小不变。现在的问题是如何处理最右 3 位 GRS。IEEE 754 定义了以下 4 种舍入方式。

1) 就近舍入：如果 $GRS > 100$，在 23 位尾数的最右位加 1；如果 $GRS < 100$，简单地把 GRS 扔掉；如果 $GRS = 100$，这时要看 23 位的尾数的最右位，当它为 1 时，要加 1，为 0 时，扔掉 GRS，保证舍入后的尾数的最右位总是 0。
2) 向下舍入：向 $-\infty$ 方向舍入，即如果结果为负并且 $GRS \neq 000$，尾数加 1。

3) 向上舍入：向 $+\infty$ 方向舍入，即如果结果为正并且 GRS $\neq$ 000，尾数加 1。

4) 向 0 舍入：最简单，扔掉 GRS。

绝大多数的 FPU 设置默认的舍入方式为就近舍入。如果我们也对上述计算结果进行就近舍入，则要在尾数 01.11101011111111111111110 的最右位加 1，变成 01.11101011111111111111111。现在我们终于得到了最后的用 IEEE 754 格式表示的结果：

C: 1 01111100 11101011111111111111111 (十六进制：BE75FFFF)

那么如何在自己的 C 语言程序中选择舍入方式呢？如前所述，在 x86 系列的 FPU 中有一个 16 位的浮点控制寄存器，其中的第 11 和 10 两位定义了 FPU 操作时的舍入方式。以下是一段测试浮点加减法及舍入方式的 C 语言程序，运行在 Linux 操作系统下，由 gcc 编译。其中对浮点控制寄存器的写入由 x86 汇编指令 fldcw 完成。数据的输入及输出均是十六进制数，程序内部完成对输入数据在不同的舍入方式下的浮点加减运算。

```
// Rounding test for (x86 + Linux + gcc)
#include<stdio.h>
#define Near asm volatile("fldcw _RoundNear")
#define Down asm volatile("fldcw _RoundDown")
#define Up   asm volatile("fldcw _RoundUp")
#define Chop asm volatile("fldcw _RoundChop")
int _RoundNear = 0x103f; // 舍入控制码=00，就近舍入
int _RoundDown = 0x143f; // 舍入控制码=01，向下舍入
int _RoundUp   = 0x183f; // 舍入控制码=10，向上舍入
int _RoundChop = 0x1c3f; // 舍入控制码=11，向 0 舍入

int main(void) {
    union {
        int intword;
        float floatword;
    } u, v, s, t;
    while (1) {
        fprintf (stderr,"input 1st fp number in hex format: ");
        fscanf (stdin,"%x",&u.intword);
        fprintf (stderr,"input 2nd fp number in hex format: ");
        fscanf (stdin,"%x",&v.intword);
        Near; // 就近舍入
        s.floatword = u.floatword + v.floatword;
        t.floatword = u.floatword - v.floatword;
        fprintf (stderr,"the sum of 2 fp numbers is (near): "
                 "%08X\t%08X\n",s.intword,t.intword);
        Down; // 向下舍入
        s.floatword = u.floatword + v.floatword;
        t.floatword = u.floatword - v.floatword;
```

```
        fprintf (stderr,"the sum of 2 fp numbers is (down): "
                 "%08X\t%08X\n",s.intword,t.intword);
        Up;   // 向上舍入
        s.floatword = u.floatword + v.floatword;
        t.floatword = u.floatword - v.floatword;
        fprintf (stderr,"the sum of 2 fp numbers is ( up ): "
                 "%08X\t%08X\n",s.intword,t.intword);
        Chop; // 向 0 舍入
        s.floatword = u.floatword + v.floatword;
        t.floatword = u.floatword - v.floatword;
        fprintf (stderr,"the sum of 2 fp numbers is (chop): "
                 "%08X\t%08X\n",s.intword,t.intword);
    }
}
```

编译命令及运行结果如下所示。输出的第一个十六进制数是相加的结果，第二个数是相减的结果。你看，舍入方式不同，结果也不同。这个程序可以被用来验证我们设计的浮点加法器是否正确 (假设 x86 FPU 的输出结果是正确的)。

```
[yamin@localhost cpu]$ gcc fadd_test.c -o fadd_test
[yamin@localhost cpu]$ fadd_test
input 1st fp number in hex format: 3C600011
input 2nd fp number in hex format: BE820000
the sum of 2 fp numbers is (near): BE75FFFF     3E890001
the sum of 2 fp numbers is (down): BE75FFFF     3E890000
the sum of 2 fp numbers is ( up ): BE75FFFE     3E890001
the sum of 2 fp numbers is (chop): BE75FFFE     3E890000
input 1st fp number in hex format:
[yamin@localhost cpu]$
```

9.3.2 浮点加法器 Verilog HDL 代码

浮点加法器的总体结构如图 9.6 所示，由三部分组成：阶码对齐，计算和规格化。以下我们描述如何用 Verilog HDL 来设计浮点加法器。模块名为 fadder，输入信号 a 和 b 是两个单精度浮点数，输入信号 sub 为 1 时 $a-b$，为 0 时 $a+b$，rm 是两位舍入控制码，输出信号 s 是单精度浮点结果。

```
module fadder (a,b,sub,rm,s);
    input [31:0] a,b; // fp inputs a and b
    input sub;        // 1: sub; 0: add
    input [1:0]  rm;  // round mode
    output [31:0] s;  // fp output
```

由于实际的操作为减法时，要从大的绝对值中减去小的绝对值，因此我们要区分出 a 和 b 哪个绝对值大。绝对值大的浮点数用 fp_large 来表示，绝对值小的用 fp_small 表示。即，如果 b 的绝对值大于 a 的绝对值，a 和 b 两个数要交换位置。

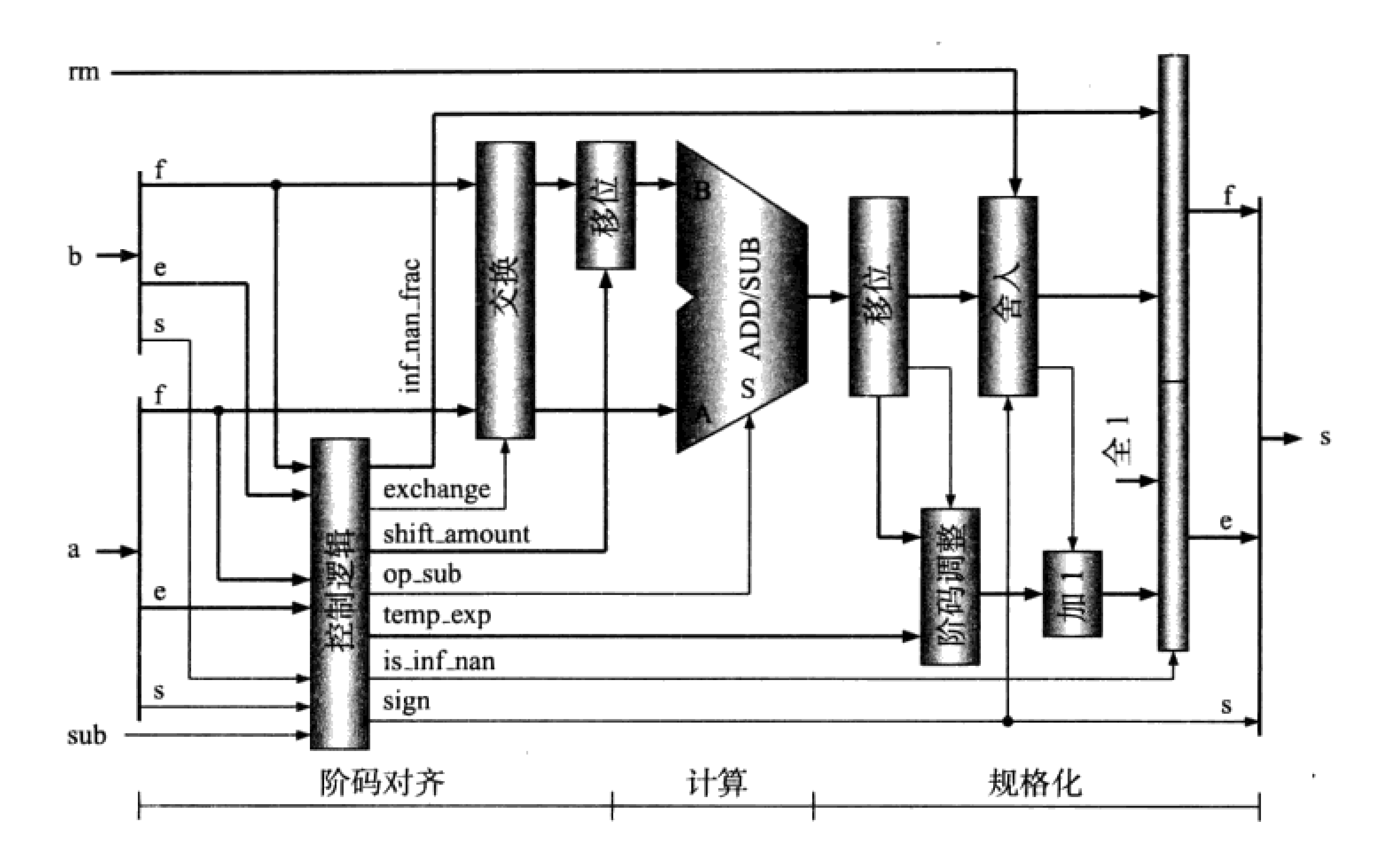


图 9.6　浮点加法器电路的总体模块图

```
wire exchange = ({1'b0,b[30:0]} > {1'b0,a[30:0]});
wire [31:0] fp_large = exchange? b : a;
wire [31:0] fp_small = exchange? a : b;
```

以下两行确定两个浮点数的隐藏位。如果阶码部分为 0，该浮点数或者为 0 或者为非规格化数。这两种情况中的隐藏位均为 0。如果是规格化数，阶码部分非 0。因此各位相或的结果就是它的隐藏位。

```
wire fp_large_hidden_bit = |fp_large[30:23];
wire fp_small_hidden_bit = |fp_small[30:23];
```

由此，我们可以得到加入了隐藏位的两个尾数 large_frac24 和 small_frac24。每个都有 24 位：1 位隐藏位和浮点数格式中的 23 位尾数。结果的阶码暂时定为与绝对值大的阶码相同的值，在对结果规格化时，我们还要调整它。现在考虑结果符号位 sign。如果 a 和 b 两个数没有交换位置，即 a 的绝对值大，计算结果的符号与 a 相同。如果交换了，即 b 的绝对值大。这时，如果是 a + b，则计算结果的符号与 b 相同；如果是 a − b，则计算结果的符号与 b 相反。真正的操作 op_sub 取决于 a 和 b 的符号位及 sub。如果 a 和 b 的符号相反并且 sub 为 0，则相减；如果 a 和 b 的符号相同并且 sub 为 1，也相减；其他情况皆相加。

```
wire [23:0] large_frac24 = {fp_large_hidden_bit,fp_large[22:0]};
wire [23:0] small_frac24 = {fp_small_hidden_bit,fp_small[22:0]};
wire [7:0] temp_exp = fp_large[30:23];
wire sign = exchange? sub ^ b[31] : a[31];
wire op_sub = sub ^ fp_large[31] ^ fp_small[31];
```

以下的代码确定两个浮点数是否为无穷大 (inf) 或者 NaN (nan)。如果阶码为全 1 并且尾数为 0，则该浮点数为无穷大；如果阶码为全 1 但尾数不为 0，则该浮点数为 NaN。信号 is_inf_nan 为 1 时，表示最后结果为无穷大或者 NaN。

```
wire fp_large_expo_is_ff =  &fp_large[30:23]; // exp == 0xff
wire fp_small_expo_is_ff =  &fp_small[30:23];
wire fp_large_frac_is_00 = ~|fp_large[22:0];  // frac == 0x0
wire fp_small_frac_is_00 = ~|fp_small[22:0];
wire fp_large_is_inf = fp_large_expo_is_ff& fp_large_frac_is_00;
wire fp_small_is_inf = fp_small_expo_is_ff& fp_small_frac_is_00;
wire fp_large_is_nan = fp_large_expo_is_ff&~fp_large_frac_is_00;
wire fp_small_is_nan = fp_small_expo_is_ff&~fp_small_frac_is_00;
wire is_inf_nan = fp_large_is_inf | fp_small_is_inf |
                  fp_large_is_nan | fp_small_is_nan;
```

只要有一个数为 NaN，结果是 NaN；两个数都不是 NaN，结果也可能是 NaN。表 9.2 给出了两个无穷大数的操作结果。

表 9.2　两个无穷大数的操作结果

sub	fp_large	fp_small	s	注 释
0	$+\infty$	$+\infty$	$+\infty$	$(+\infty)+(+\infty)$
0	$-\infty$	$-\infty$	$-\infty$	$(-\infty)+(-\infty)$
1	$+\infty$	$-\infty$	$+\infty$	$(+\infty)-(-\infty)$
1	$-\infty$	$+\infty$	$-\infty$	$(-\infty)-(+\infty)$
0	$+\infty$	$-\infty$	NaN	$(+\infty)+(-\infty)$
0	$-\infty$	$+\infty$	NaN	$(-\infty)+(+\infty)$
1	$+\infty$	$+\infty$	NaN	$(+\infty)-(+\infty)$
1	$-\infty$	$-\infty$	NaN	$(-\infty)-(-\infty)$

当两个数都不是 NaN，检查表 9.2 中输出为 NaN 的条件。如果满足，设置结果为 NaN。否则，结果是无穷大。因为结果不管是无穷大还是 NaN，它们的阶码都是全 1，所以我们只设置尾数部分 (inf_nan_frac)。当结果为 NaN 时，尾数部分选择 a 和 b 中较大的那个尾数。

```
wire s_is_nan = fp_large_is_nan | fp_small_is_nan |
               ((sub ^ fp_small[31] ^ fp_large[31]) &
                fp_large_is_inf & fp_small_is_inf);
wire [22:0] nan_frac = ({1'b0,a[22:0]} > {1'b0,b[22:0]}) ?
                        {1'b1,a[21:0]} : {1'b1,b[21:0]};
wire [22:0] inf_nan_frac = s_is_nan? nan_frac : 23'h0;
```

以下的代码对 small_frac24 右移并计算尾数结果。如果两个数都是规格化数，右移位数 shift_amount 等于两个数阶码的差值 exp_diff。如果 fp_large 是规格化数而

fp_small 是非规格化数，则右移位数等于 exp_diff − 1。这是因为规格化数的绝对值等于 $2^{e-127} \times 1.f$，而非规格化数的绝对值等于 $2^{0-126} \times 0.f$。图 9.7 说明当右移位数大约 26 时，只要右移 26 位即可。在做计算时，要在最左边多加一位，因为尾数相加的结果可能大于或等于 2。

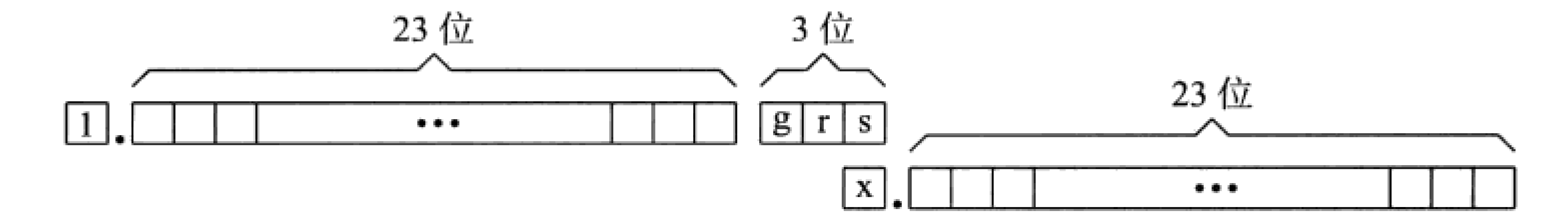


图 9.7 最多把 small_frac24 右移 26 位

```
wire [7:0] exp_diff = fp_large[30:23] - fp_small[30:23];
wire small_den_only = (fp_large[30:23] != 0) & (fp_small[30:23] == 0);
wire [7:0] shift_amount = small_den_only? exp_diff - 8'h1 : exp_diff;
wire [49:0] small_frac50 = (shift_amount >= 26)?
                           {26'h0,small_frac24} :
                           {small_frac24,26'h0} >> shift_amount;
wire [26:0] small_frac27 = {small_frac50[49:24],|small_frac50[23:0]};
wire [27:0] aligned_large_frac = {1'b0,large_frac24,3'b000};
wire [27:0] aligned_small_frac = {1'b0,small_frac27};
wire [27:0] cal_frac = op_sub?
                       aligned_large_frac - aligned_small_frac :
                       aligned_large_frac + aligned_small_frac;
```

现在开始对计算出的尾数 cal_frac 规格化。我们对以下两种情况分别加以考虑：一种是 cal_frac = 1x.xxx· · ·；另一种是 cal_frac = 0x.xxx· · ·。第一种情况比较简单，只要把 cal_frac 右移一位，再把阶码加 1。第二种情况比较复杂，要从左边开始找出第一个 1 所在的位置，或者数出第一个 1 的左边有多少个 0 (不包括 cal_frac 最左边的 0)。例如，如果 cal_frac = 00.001xxxxxxxxxxxxxxxxxxx xxx，0 的个数为 3 (代码中的 zeros)。即，要把 cal_frac 左移 3 位，然后从阶码中减去 3。阶码有可能小于 zeros，这时要把结果设置成非规格化数。

```
wire [26:0] f4,f3,f2,f1,f0;
wire [4:0] zeros;
assign    zeros[4] = ~|cal_frac[26:11];          // 16-bit 0
assign    f4 = zeros[4]? {cal_frac[10:0],16'b0} : cal_frac[26:0];
assign    zeros[3] = ~|f4[26:19];                 //  8-bit 0
assign    f3 = zeros[3]? {f4[18:0], 8'b0} : f4;
assign    zeros[2] = ~|f3[26:23];                 //  4-bit 0
assign    f2 = zeros[2]? {f3[22:0], 4'b0} : f3;
assign    zeros[1] = ~|f2[26:25];                 //  2-bit 0
assign    f1 = zeros[1]? {f2[24:0], 2'b0} : f2;
assign    zeros[0] = ~f1[26];                     //  1-bit 0
assign    f0 = zeros[0]? {f1[25:0], 1'b0} : f1;
```

```
reg [7:0]     exp0;
reg [26:0]    frac0;
always @ * begin
    if (cal_frac[27]) begin
        frac0 = cal_frac[27:1];        // 1x.xxxxxxxxxxxxxxxxxxxxxxx xxx
        exp0 = temp_exp + 8'h1;
    end else begin
        if ((temp_exp > zeros) && (f0[26])) begin
            exp0 = temp_exp - zeros; // 01.xxxxxxxxxxxxxxxxxxxxxxx xxx
            frac0 = f0;
        end else begin
            exp0 = 0;                  // is a denormalized number or 0
            if (temp_exp != 0)         // (e - 127) = ((e - 1) - 126)
                frac0 = cal_frac[26:0] << (temp_exp - 8'h1);
            else frac0 = cal_frac[26:0];
        end
    end
end
```

注意，exp0 可能变成全 1，我们将在下面讨论它。表 9.3 列出了舍入时尾数必须加 1 的所有情况，frac0[2:0] 就是 GRS。如果 frac_plus_1 为 1，尾数加 1。

表 9.3　舍入操作

rm[1:0]	frac0[3:0]	sign	frac_plus_1	注 释
0 0	1 1 0 0	x	1	就近舍入
0 0	x 1 非00	x	1	就近舍入
0 1	x 非000	1	1	向下舍入
1 0	x 非000	0	1	向上舍入

```
wire frac_plus_1 =
    ~rm[1] & ~rm[0] &  frac0[2] & (frac0[1] |  frac0[0]) |
    ~rm[1] & ~rm[0] &  frac0[2] & ~frac0[1] & ~frac0[0]  &  frac0[3] |
    ~rm[1] &  rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) &  sign |
     rm[1] & ~rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) & ~sign;
wire [24:0] frac_round = {1'b0,frac0[26:3]} + frac_plus_1;
```

如果加 1 之前的尾数为全 1，加 1 之后 frac_round 等于 2，如下所示。这时还要把阶码加 1，尾数右移一位。

```
  0 1.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
+ 0 0.0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
-----------------------------------------------------
= 1 0.0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

阶码调整后可能变为全 1，还有前面讲过的 exp0 也可能变为全 1。IEEE 754 称其为上溢。IEEE 754 对上溢的处理规定如下。

1) 就近舍入时把上溢结果置成无穷大。
2) 向 0 舍入时把上溢结果置成最大的规格化数。
3) 向下舍入时把负数的上溢结果置成无穷大；把正数的上溢结果置成绝对值最大的正规格化数。
4) 向上舍入时把正数的上溢结果置成无穷大；把负数的上溢结果置成绝对值最大的负规格化数。

举例如下。计算 C = A + B。

```
A = 0 11111110 11111111111111111111111
B = 0 11111011 11111111111111111111111
```

```
    0 1.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 000
+   0 0.0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111
=   1 0.0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 111
→   0 1.0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 011
```

尾数右移一位，阶码加 1，变为全 1。这时尾数没有用了，结果 C 有两种可能：

```
C = 0 11111111 00000000000000000000000 (就近或向上舍入)
C = 0 11111110 11111111111111111111111 (向 0 或向下舍入)
```

我们使用 fuction 实现对上溢的处理，其中的 casex 语句允许使用任意值 “x” 。

```
wire [7:0]  exponent = frac_round[24]? exp0 + 8'h1 : exp0;
wire overflow = &exp0 | &exponent;
wire [7:0]  final_exponent;
wire [22:0] final_fraction;
assign {final_exponent,final_fraction} = final_result(overflow, rm,
        sign, is_inf_nan, exponent, frac_round[22:0], inf_nan_frac);
assign s = {sign,final_exponent,final_fraction};

function [30:0]  final_result;
    input         overflow;
    input [1:0]   rm;
    input         sign, is_inf_nan;
    input [7:0]   exponent;
    input [22:0]  fraction, inf_nan_frac;
    casex ({overflow, rm, sign, is_inf_nan})
        5'b1_00_x_x : final_result = {8'hff,23'h000000};   // inf
        5'b1_01_0_x : final_result = {8'hfe,23'h7fffff};   // max
        5'b1_01_1_x : final_result = {8'hff,23'h000000};   // inf
        5'b1_10_0_x : final_result = {8'hff,23'h000000};   // inf
        5'b1_10_1_x : final_result = {8'hfe,23'h7fffff};   // max
        5'b1_11_x_x : final_result = {8'hfe,23'h7fffff};   // max
        5'b0_xx_x_0 : final_result = {exponent,fraction};  // normal
```

```
            5'b0_xx_x_1 : final_result = {8'hff,inf_nan_frac}; // inf_nan
            default     : final_result = {8'h00,23'h000000};   // 0
        endcase
    endfunction
endmodule
```

整个代码到此结束。以下给出几个仿真结果，以验证上述代码的正确性。图 9.8 示出的是本节开始提到的两个规格化数的加减操作。从图中我们可以看出舍入方式对结果尾数的影响。

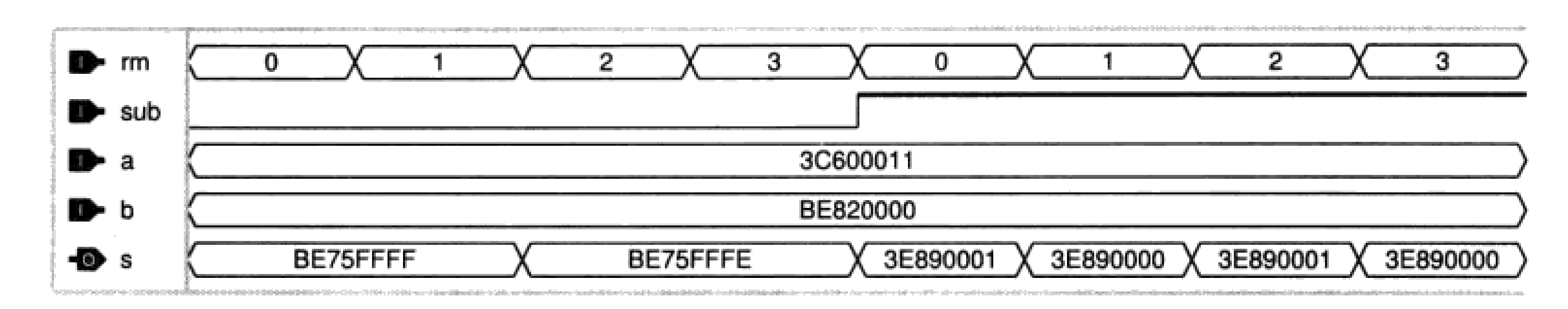


图 9.8　浮点加法器仿真结果: 舍入控制对尾数的影响

选择不同的舍入方式不但对结果尾数有影响，对阶码也有影响，见图 9.9。加法结果依舍入方式的不同，有 `40000000` 和 `3FFFFFFF` 两种结果，它们的阶码不同。

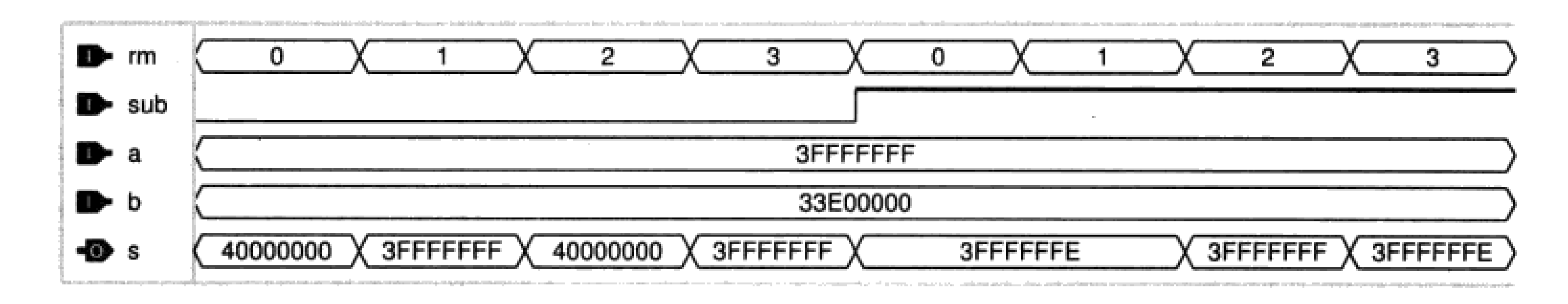


图 9.9　浮点加法器仿真结果: 舍入控制对阶码的影响

图 9.10 给出了在就近舍入方式下 8 种特殊的计算。第 1 个是正无穷大加正无穷大，结果为无穷大。第 2 个是正无穷大减正无穷大，结果为 NaN。第 3 个是规格化数加 NaN，结果为 NaN。第 4 个是两个绝对值最大的规格化数相加，结果为无穷大。第 5 个是规格化数加 0。第 6 个是绝对值最小的规格化数加上绝对值最大的非规格化数。最后两个是两个非规格化数的加减。

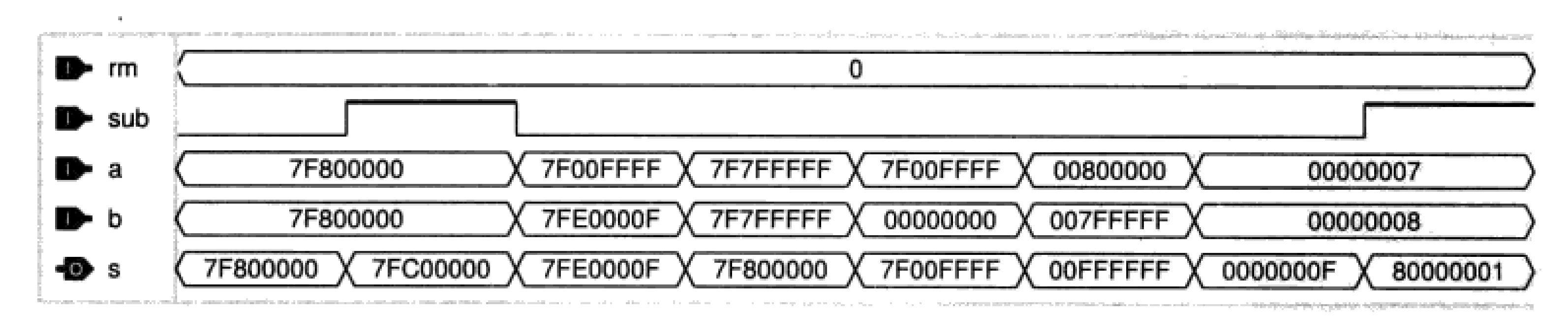


图 9.10　浮点加法器仿真结果: 8 种特殊的计算

9.3.3 流水线浮点加法器设计

流水线浮点加法器每个周期都可以接收一条浮点加法或减法指令，总体结构见图 9.11。为了叙述简便，我们把所有的计算分为 3 级：阶码对齐级，计算级和规格化级。注意在实际的 FPU 设计中，流水线分级的规则是使每一级的操作花费大致相同的时间。相邻两级之间需要加入流水线寄存器。图中的信号名称有如下规则：阶码对齐级的信号都加头文字 a，计算级的信号都加 c，规格化级的信号都加 n。

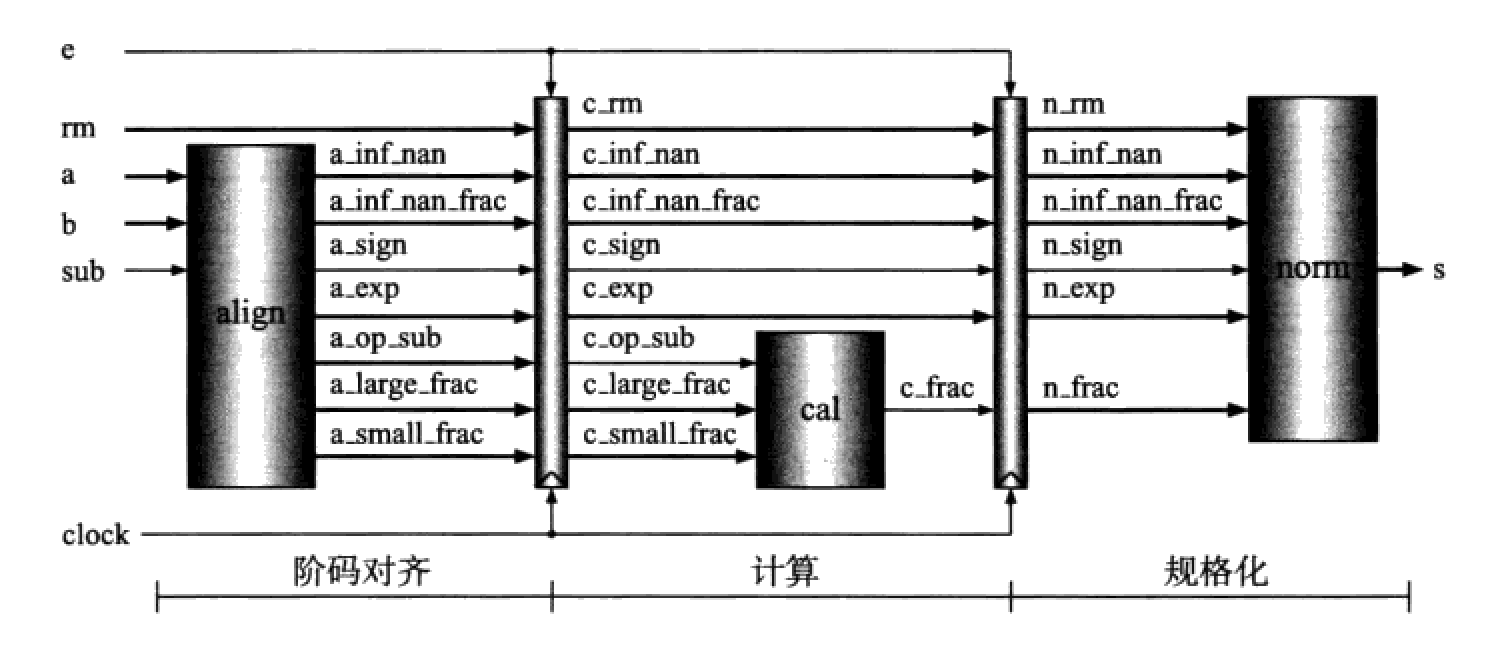


图 9.11 流水线浮点加法器总体结构图

以下是流水线浮点加法器的最顶层代码。它只是给出各模块之间的信号连接关系。总共有 5 个模块，它们分别是：(1) fadd_align (阶码对齐模块)；(2) reg_align_cal (阶码对齐级与计算级之间的寄存器模块)；(3) fadd_cal (计算模块)；(4) reg_cal_norm (计算级与规格化级之间的寄存器模块)；(5) fadd_norm (规格化模块)。全部模块的详细代码在下面给出。

```
module pipelined_fadder (a,b,sub,rm,s,clock,clrn,e);
    input [31:0] a,b; // fp inputs a and b
    input        sub; // 1: sub; 0: add
    input        e;   // enable
    input [1:0]  rm;  // round mode
    input        clock,clrn;
    output [31:0] s;  // fp output
    //alignment stage:
    wire [1:0]    a_rm;
    wire          a_is_inf_nan;
    wire [22:0]   a_inf_nan_frac;
    wire          a_sign;
    wire [7:0]    a_exp;
    wire          a_op_sub;
    wire [23:0]   a_large_frac;
    wire [26:0]   a_small_frac;
    fadd_align alignment (a,b,sub,a_is_inf_nan,a_inf_nan_frac,a_sign,
```

```
                           a_exp,a_op_sub,a_large_frac,a_small_frac);
    // pipelined registers between alignment and calculation:
    wire [1:0]     c_rm;
    wire           c_is_inf_nan;
    wire [22:0]    c_inf_nan_frac;
    wire           c_sign;
    wire [7:0]     c_exp;
    wire           c_op_sub;
    wire [23:0]    c_large_frac;
    wire [26:0]    c_small_frac;
    reg_align_cal reg_ac (rm,a_is_inf_nan,a_inf_nan_frac,a_sign,a_exp,
                          a_op_sub,a_large_frac,a_small_frac,clock,clrn,
                          e,c_rm,c_is_inf_nan,c_inf_nan_frac,c_sign,
                          c_exp,c_op_sub,c_large_frac,c_small_frac);
    // calculation stage:
    wire [27:0]    c_frac;
    fadd_cal calculation (c_op_sub,c_large_frac,c_small_frac,c_frac);
    // pipelined registers between calculation and normalization:
    wire [1:0]     n_rm;
    wire           n_is_inf_nan;
    wire [22:0]    n_inf_nan_frac;
    wire           n_sign;
    wire [7:0]     n_exp;
    wire [27:0]    n_frac;
    reg_cal_norm reg_cn (c_rm,c_is_inf_nan,c_inf_nan_frac,c_sign,c_exp,
                         c_frac,clock,clrn,e, n_rm,n_is_inf_nan,
                         n_inf_nan_frac,n_sign,n_exp,n_frac);
    // normalization stage:
    fadd_norm normalization (n_rm,n_is_inf_nan,n_inf_nan_frac,
                             n_sign,n_exp,n_frac,s);
endmodule
```

以下是阶码对齐级的代码。它与 9.3.2 小节描述的浮点加法器中的代码类似，只是此处代码中的输出信号可能是 9.3.2 小节代码中的内部信号。如果是这种情况，我们把原来的表示内部信号的关键字 wire 改成了关键字 assign。

```
module fadd_align
  (a,b,sub,
   is_inf_nan,inf_nan_frac,sign,temp_exp,op_sub,large_frac24,small_frac27);
    input [31:0] a,b;
    input        sub;
    output       is_inf_nan;
    output [22:0] inf_nan_frac;
    output        sign;
    output [7:0]  temp_exp;
    output        op_sub;
    output [23:0] large_frac24;
    output [26:0] small_frac27;
```

```
    wire        exchange = ({1'b0,b[30:0]} > {1'b0,a[30:0]});
    wire [31:0] fp_large = exchange? b : a;
    wire [31:0] fp_small = exchange? a : b;
    wire        fp_large_hidden_bit = |fp_large[30:23];
    wire        fp_small_hidden_bit = |fp_small[30:23];
    wire [23:0] large_frac24 = {fp_large_hidden_bit,fp_large[22:0]};
    wire [23:0] small_frac24 = {fp_small_hidden_bit,fp_small[22:0]};
    assign temp_exp = fp_large[30:23];
    assign sign = exchange? sub ^ b[31] : a[31];
    assign op_sub = sub ^ fp_large[31] ^ fp_small[31];
    wire   fp_large_expo_is_ff =  &fp_large[30:23]; // exp == 0xff
    wire   fp_small_expo_is_ff =  &fp_small[30:23];
    wire   fp_large_frac_is_00 = ~|fp_large[22:0];  // frac == 0x0
    wire   fp_small_frac_is_00 = ~|fp_small[22:0];
    wire   fp_large_is_inf = fp_large_expo_is_ff &  fp_large_frac_is_00;
    wire   fp_small_is_inf = fp_small_expo_is_ff &  fp_small_frac_is_00;
    wire   fp_large_is_nan = fp_large_expo_is_ff & ~fp_large_frac_is_00;
    wire   fp_small_is_nan = fp_small_expo_is_ff & ~fp_small_frac_is_00;
    assign is_inf_nan = fp_large_is_inf | fp_small_is_inf |
                        fp_large_is_nan | fp_small_is_nan;
    wire   s_is_nan   =  fp_large_is_nan | fp_small_is_nan |
                        ((sub ^ fp_small[31] ^ fp_large[31]) &
                        fp_large_is_inf & fp_small_is_inf);
    wire [22:0] nan_frac = ({1'b0,a[22:0]} > {1'b0,b[22:0]}) ?
                           {1'b1,a[21:0]} : {1'b1,b[21:0]};
    assign      inf_nan_frac = s_is_nan? nan_frac : 23'h0;
    wire [7:0]  exp_diff = fp_large[30:23] - fp_small[30:23];
    wire small_den_only = (fp_large[30:23] != 0) & (fp_small[30:23] == 0);
    wire [7:0]  shift_amount = small_den_only? exp_diff - 8'h1 : exp_diff;
    wire [49:0] small_frac50 = (shift_amount >= 26)?
                {26'h0,small_frac24} :
                {small_frac24,26'h0} >> shift_amount;
    assign      small_frac27 = {small_frac50[49:24],|small_frac50[23:0]};
endmodule
```

以下是阶码对齐级与计算级之间的寄存器模块的代码。

```
module reg_align_cal (a_rm,a_is_inf_nan,a_inf_nan_frac,a_sign,a_exp,
                      a_op_sub,a_large_frac,a_small_frac,clock,clrn,
                      e,c_rm,c_is_inf_nan,c_inf_nan_frac,c_sign,
                      c_exp,c_op_sub,c_large_frac,c_small_frac);
    input        e;   // enable
    input [1:0]  a_rm;
    input        a_is_inf_nan;
    input [22:0] a_inf_nan_frac;
    input        a_sign;
    input [7:0]  a_exp;
    input        a_op_sub;
```

```
    input  [23:0] a_large_frac;
    input  [26:0] a_small_frac;
    input         clock,clrn;
    output [1:0] c_rm;
    output        c_is_inf_nan;
    output [22:0] c_inf_nan_frac;
    output         c_sign;
    output [7:0]  c_exp;
    output         c_op_sub;
    output [23:0] c_large_frac;
    output [26:0] c_small_frac;
    reg [1:0]      c_rm;
    reg            c_is_inf_nan;
    reg [22:0]     c_inf_nan_frac;
    reg            c_sign;
    reg [7:0]      c_exp;
    reg            c_op_sub;
    reg [23:0]     c_large_frac;
    reg [26:0]     c_small_frac;
    always @ (posedge clock or negedge clrn) begin
        if (clrn == 0) begin
            c_rm           <= 0;
            c_is_inf_nan   <= 0;
            c_inf_nan_frac <= 0;
            c_sign         <= 0;
            c_exp          <= 0;
            c_op_sub       <= 0;
            c_large_frac   <= 0;
            c_small_frac   <= 0;
        end else if (e) begin
            c_rm           <= a_rm;
            c_is_inf_nan   <= a_is_inf_nan;
            c_inf_nan_frac <= a_inf_nan_frac;
            c_sign         <= a_sign;
            c_exp          <= a_exp;
            c_op_sub       <= a_op_sub;
            c_large_frac   <= a_large_frac;
            c_small_frac   <= a_small_frac;
        end
    end
endmodule
```

计算级模块的代码如下。我们在本模块中加入了加法器两个输入的最高位 0 以及绝对值大的尾数右边的 GRS 位。这样可以节省 5 个 D 触发器，因为它们的值永远为 0。同样，我们用 assign 指定 cal_frac 的输出值。

```
module fadd_cal (op_sub,large_frac24,small_frac27, cal_frac);
    input op_sub;
    input [23:0] large_frac24;
    input [26:0] small_frac27;
    output [27:0] cal_frac;
    wire [27:0]   aligned_large_frac = {1'b0,large_frac24,3'b000};
    wire [27:0]   aligned_small_frac = {1'b0,small_frac27};
    assign        cal_frac = op_sub?
                          aligned_large_frac - aligned_small_frac :
                          aligned_large_frac + aligned_small_frac;
endmodule
```

以下的代码描述计算级与规格化级之间的寄存器模块。

```
module reg_cal_norm
  (c_rm,c_is_inf_nan,c_inf_nan_frac,c_sign,c_exp,c_frac,clock,clrn,
   e,n_rm,n_is_inf_nan,n_inf_nan_frac,n_sign,n_exp,n_frac);
    input        e;   // enable
    input [1:0] c_rm;
    input        c_is_inf_nan;
    input [22:0] c_inf_nan_frac;
    input         c_sign;
    input [7:0]  c_exp;
    input [27:0] c_frac;
    input         clock,clrn;
    output [1:0] n_rm;
    output        n_is_inf_nan;
    output [22:0] n_inf_nan_frac;
    output         n_sign;
    output [7:0]  n_exp;
    output [27:0] n_frac;
    reg [1:0]     n_rm;
    reg           n_is_inf_nan;
    reg [22:0]    n_inf_nan_frac;
    reg           n_sign;
    reg [7:0]     n_exp;
    reg [27:0]    n_frac;
    always @ (posedge clock or negedge clrn) begin
        if (clrn == 0) begin
            n_rm           <= 0;
            n_is_inf_nan   <= 0;
            n_inf_nan_frac <= 0;
            n_sign         <= 0;
            n_exp          <= 0;
            n_frac         <= 0;
        end else if (e) begin
```

```
            n_rm            <= c_rm;
            n_is_inf_nan    <= c_is_inf_nan;
            n_inf_nan_frac  <= c_inf_nan_frac;
            n_sign          <= c_sign;
            n_exp           <= c_exp;
            n_frac          <= c_frac;
        end
    end
endmodule
```

最后一个模块完成规格化操作。它也是与第一节所给出的代码类似。其中信号 s 为流水线浮点加法器的最终输出。

```
module fadd_norm (rm,is_inf_nan,inf_nan_frac,sign,temp_exp,cal_frac,s);
    input [1:0] rm;
    input       is_inf_nan;
    input [22:0] inf_nan_frac;
    input        sign;
    input [7:0]  temp_exp;
    input [27:0] cal_frac;
    output [31:0] s;
    wire [26:0]   f4,f3,f2,f1,f0;
    wire [4:0]    zeros;
    assign        zeros[4] = ~|cal_frac[26:11];          // 16-bit 0
    assign        f4 = zeros[4]? {cal_frac[10:0],16'b0} : cal_frac[26:0];
    assign        zeros[3] = ~|f4[26:19];                // 8-bit 0
    assign        f3 = zeros[3]? {f4[18:0], 8'b0} : f4;
    assign        zeros[2] = ~|f3[26:23];                // 4-bit 0
    assign        f2 = zeros[2]? {f3[22:0], 4'b0} : f3;
    assign        zeros[1] = ~|f2[26:25];                // 2-bit 0
    assign        f1 = zeros[1]? {f2[24:0], 2'b0} : f2;
    assign        zeros[0] = ~f1[26];                    // 1-bit 0
    assign        f0 = zeros[0]? {f1[25:0], 1'b0} : f1;
    reg [7:0]     exp0;
    reg [26:0]    frac0;
    always @ * begin
        if (cal_frac[27]) begin
            frac0 = cal_frac[27:1];      // 1x.xxxxxxxxxxxxxxxxxxxxxx xxx
            exp0 = temp_exp + 8'h1;
        end else begin
            if ((temp_exp > zeros) && (f0[26])) begin
                exp0 = temp_exp - zeros; // 01.xxxxxxxxxxxxxxxxxxxxxx xxx
                frac0 = f0;
            end else begin
                exp0 = 0;                // is a denormalized number or 0
                if (temp_exp != 0)       // (e - 127) = ((e - 1) - 126)
                    frac0 = cal_frac[26:0] << (temp_exp - 8'h1);
                else frac0 = cal_frac[26:0];
```

```
            end
        end
    end
    wire frac_plus_1 =
         ~rm[1] & ~rm[0] &  frac0[2] & (frac0[1] |  frac0[0]) |
         ~rm[1] & ~rm[0] &  frac0[2] & ~frac0[1] & ~frac0[0]  &  frac0[3] |
         ~rm[1] &  rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) &  sign |
          rm[1] & ~rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) & ~sign;
    wire [24:0] frac_round = {1'b0,frac0[26:3]} + frac_plus_1;
    wire [7:0]  exponent = frac_round[24]? exp0 + 8'h1 : exp0;
    wire overflow = &exp0 | &exponent;
    wire [7:0]  final_exponent;
    wire [22:0] final_fraction;
    assign {final_exponent,final_fraction} = final_result(overflow, rm,
            sign, is_inf_nan, exponent, frac_round[22:0], inf_nan_frac);
    assign s = {sign,final_exponent,final_fraction};
    function [30:0]  final_result;
        input        overflow;
        input [1:0]  rm;
        input        sign, is_inf_nan;
        input [7:0]  exponent;
        input [22:0] fraction, inf_nan_frac;
        casex ({overflow, rm, sign, is_inf_nan})
            5'b1_00_x_x : final_result = {8'hff,23'h000000};   // inf
            5'b1_01_0_x : final_result = {8'hfe,23'h7fffff};   // max
            5'b1_01_1_x : final_result = {8'hff,23'h000000};   // inf
            5'b1_10_0_x : final_result = {8'hff,23'h000000};   // inf
            5'b1_10_1_x : final_result = {8'hfe,23'h7fffff};   // max
            5'b1_11_x_x : final_result = {8'hfe,23'h7fffff};   // max
            5'b0_xx_x_0 : final_result = {exponent,fraction};  // normal
            5'b0_xx_x_1 : final_result = {8'hff,inf_nan_frac}; // inf_nan
            default     : final_result = {8'h00,23'h000000};   // 0
        endcase
    endfunction
endmodule
```

图 9.12 给出了流水线浮点加法器的仿真结果。图中左上的数字 1，2 和 3 表示浮点数 3C600011 减 BE820000 时的流水线级数。在第 3 级出来结果，即 3E890001。

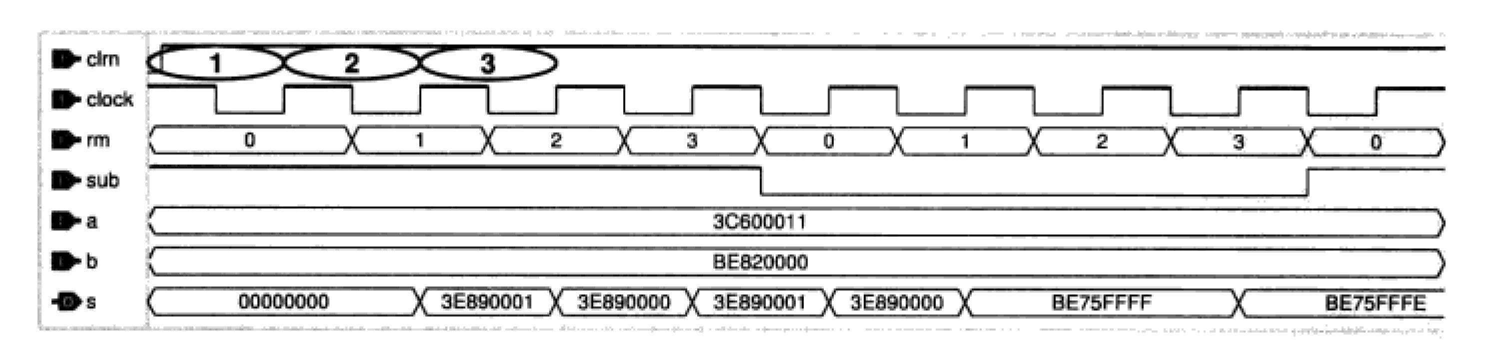


图 9.12 流水线浮点加法器仿真结果

我们将在第 10 章使用这个流水线浮点加法器来设计一个带有 FPU 的流水线 CPU。此处的 3 级流水线全部属于 CPU 的执行级。它们的前面是取指令级 IF 和指令译码级 ID，它们的后面是结果写回级 WB。即，这样的 CPU 执行一条浮点加减法指令时将花费 6 个时钟周期 (Latency)，但每个时钟周期 CPU 可以接收一条浮点加法或减法指令 (Throughput)。

9.4 浮点乘法器 FMUL 设计

本节首先讨论浮点乘法算法，然后给出用 Wallace 树型乘法算法设计的浮点乘法器的 Verilog HDL 源代码，最后描述流水线浮点乘法器的设计方法。

9.4.1 浮点乘法算法

因为没有阶码对齐的问题，浮点乘法算法比浮点加法算法要简单一些。首先考虑两个规格化浮点数相乘。设 $a = \{s_a, e_a, f_a\}$，$b = \{s_b, e_b, f_b\}$ 为两个 IEEE 754 单精度浮点数，试计算 $c = \{s_c, e_c, f_c\} = a \times b$。c 的符号 $s_c = s_a \oplus s_b$，c 的绝对值 $|c| = |a| \times |b| = (2^{e_a - 127} \times 1.f_a) \times (2^{e_b - 127} \times 1.f_b) = 2^{(e_a + e_b - 127) - 127} \times (1.f_a \times 1.f_b)$。我们有 $1.0 \leqslant (1.f_a \times 1.f_b) < 4.0$。如果 $1.f_a \times 1.f_b < 2.0$，则 $e_c = e_a + e_b - 127$，$1.f_c = 1.f_a \times 1.f_b$，否则 $e_c = e_a + e_b - 127 + 1$，$1.f_c = (1.f_a \times 1.f_b) >> 1$ (右移一位)。

因为规格化数的阶码 e 满足 $1 \leqslant e \leqslant 254$，所以 $-125 \leqslant (e_a + e_b - 127) \leqslant 381$，$-124 \leqslant (e_a + e_b - 127 + 1) \leqslant 382$。即，$e_c$ 有可能超出 $1 \sim 254$ 的范围。当 $1 \leqslant e_c \leqslant 254$ 时，相乘结果为规格化数；当 $e_c > 254$ 时，结果用无穷大 ($e_c = 255$，$f_c = 0$) 表示；当 $e_c < 1$ 时，如果相乘结果大于或等于 $2^{-126} \times 0.00000000000000000000001 = 2^{-149}$ 时，可以用非规格化数表示，否则用 0 表示。

规格化数的阶码 e 满足 $1 \leqslant e \leqslant 254$ 意味着实际的阶码 e' (即 $e - 127$) 满足 $-126 \leqslant e' \leqslant 127$。设 $a = \{s_a, e_a, f_a\}$ 为规格化数，$b = \{s_b, e_b, f_b\}$ 为非规格化数 ($e_b = 0$，$f_b \neq 0$)。$c = a \times b$ 的绝对值 $|c| = |a| \times |b| = (2^{e_a - 127} \times 1.f_a) \times (2^{-126} \times 0.f_b) = 2^{(e_a - 253)} \times (1.f_a \times 0.f_b)$。最大绝对值为 $2^{254-253} \times (2 - 2^{-23}) \times (1 - 2^{-23}) = 2 \times (2 - 3 \times 2^{-23} + 2^{-46})$，这是一个规格化数。最小绝对值为 $2^{1-253} \times 1.0 \times 2^{-23} = 2^{-275}$。这已经超出了 e' 所应在的范围，即结果可能为非规格化数或者 0。

设 $a = \{s_a, e_a, f_a\}$，$b = \{s_b, e_b, f_b\}$ 都是非规格化数。$c = a \times b$ 的绝对值 $|c| = |a| \times |b| = (2^{-126} \times 0.f_a) \times (2^{-126} \times 0.f_b) = 2^{-252} \times (0.f_a \times 0.f_b)$。结果比非规格化数所能表示的最小绝对值还要小，应设其为 0。

最后讨论几种特殊的运算：(1) 假设 $b \neq 0$ 并且 $b \neq NaN$，则 $\infty \times b = \infty$；(2) $NaN \times b = NaN$；(3) $\infty \times 0 = NaN$。

以下的程序用来测试 x86 执行浮点乘法时产生的结果 (用十六进制数表示)。该程序的执行结果对浮点乘法器的设计有参考作用。

```
// Rounding test for (x86 + Linux + gcc)
#include<stdio.h>
#define Near asm volatile("fldcw _RoundNear")
#define Down asm volatile("fldcw _RoundDown")
#define Up   asm volatile("fldcw _RoundUp")
#define Chop asm volatile("fldcw _RoundChop")
int _RoundNear = 0x103f; // 舍入控制码=00，就近舍入
int _RoundDown = 0x143f; // 舍入控制码=01，向下舍入
int _RoundUp   = 0x183f; // 舍入控制码=10，向上舍入
int _RoundChop = 0x1c3f; // 舍入控制码=11，向0舍入
int main(void) {
    union {
        int intword;
        float floatword;
    } u, v, s;
    while (1) {
        fprintf (stderr,"input 1st f_p number in hex format: ");
        fscanf (stdin,"%x",&u.intword);
        fprintf (stderr,"input 2nd f_p number in hex format: ");
        fscanf (stdin,"%x",&v.intword);
        Near; // 就近舍入
        s.floatword = u.floatword * v.floatword;
        fprintf (stderr,"the prod of 2 fp numbers is (near): "
                 "%08X\n",s.intword);
        Down; // 向下舍入
        s.floatword = u.floatword * v.floatword;
        fprintf (stderr,"the prod of 2 fp numbers is (down): "
                 "%08X\n",s.intword);
        Up;   // 向上舍入
        s.floatword = u.floatword * v.floatword;
        fprintf (stderr,"the prod of 2 fp numbers is ( up ): "
                 "%08X\n",s.intword);
        Chop; // 向0舍入
        s.floatword = u.floatword * v.floatword;
        fprintf (stderr,"the prod of 2 fp numbers is (chop): "
                 "%08X\n",s.intword);
    }
}
```

我们选择 8 组数据来运行上述程序，就近舍入的运行结果在表 9.4 中列出。第 1 组示出规格化时阶码加 1 ($1.5 \times 1.5 = 2.25$)；第 2 组是两个绝对值最小的规格化数相乘，结果为 0；第 3 组是两个绝对值最大的规格化数相乘，结果为无穷大 (就近舍入)；第 4 组表示两个规格化数相乘结果为非规格化数的情况；第 5 组是一个非规格化数乘以一个规格化数；第 6 组是无穷大乘以一个规格化数，结果为无穷大；第 7 组是无穷大乘以 0，结果为 NaN；第 8 组是 NaN 乘以一个规格化数，结果为 NaN。

表 9.4 浮点乘法操作结果

	1	2	3	4	5	6	7	8
a	3FC00000	00800000	7F7FFFFF	00800000	003FFFFF	7F800000	7F800000	7FF000FF
b	3FC00000	00800000	7F7FFFFF	3F000000	40000000	00FFFFFF	00000000	3F80FF00
s	40100000	00000000	7F800000	00400000	007FFFFE	7F800000	FFC00000	7FF000FF

9.4.2 Wallace 树型浮点乘法器 Verilog HDL 代码

浮点乘法器的总体结构如图 9.13 所示，由三部分组成：(1) 用 Wallace Tree 计算部分积；(2) 部分积相加；(3) 规格化。以下我们描述如何用 Verilog HDL 来设计浮点乘法器。模块名为 fmul，输入信号 a 和 b 是两个单精度浮点数，rm 是两位舍入控制码，输出信号 s 是单精度浮点结果。

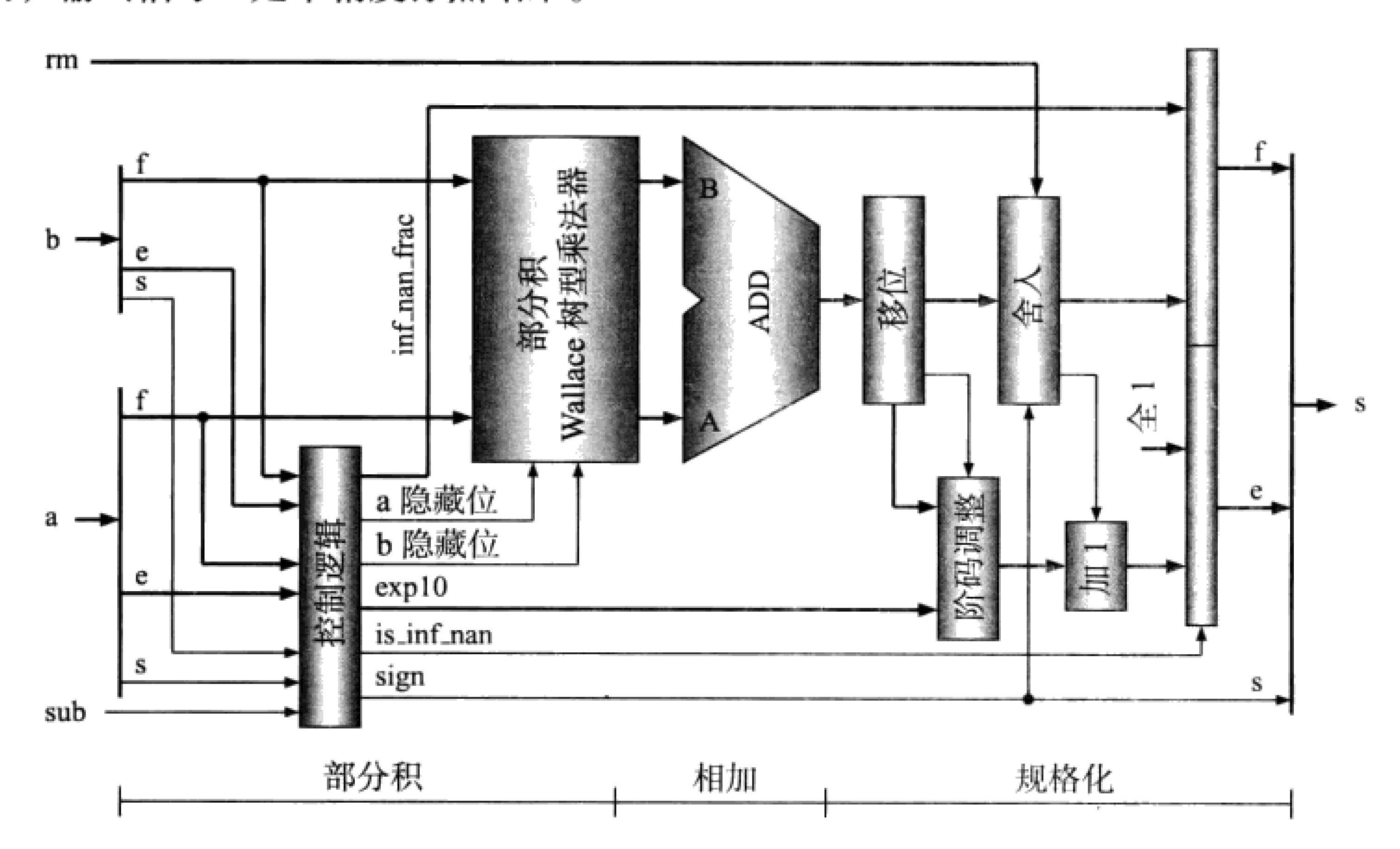


图 9.13 浮点乘法器电路的总体模块图

```
module fmul (a,b,rm,s);
    input  [31:0] a,b; // fp inputs a and b
    input  [1:0]  rm;  // round mode
    output [31:0] s;   // fp output
```

判断结果是否为无穷大或者 NaN 以及设定无穷大或者 NaN 的尾数。

```
    wire a_expo_is_00 = ~|a[30:23]; // exp = 00
    wire b_expo_is_00 = ~|b[30:23];
    wire a_expo_is_ff =  &a[30:23]; // exp = ff
    wire b_expo_is_ff =  &b[30:23];
    wire a_frac_is_00 = ~|a[22:0];  // frac = 0
```

```
wire b_frac_is_00 = ~|b[22:0];
wire a_is_inf = a_expo_is_ff &  a_frac_is_00;
wire b_is_inf = b_expo_is_ff &  b_frac_is_00;
wire a_is_nan = a_expo_is_ff & ~a_frac_is_00;
wire b_is_nan = b_expo_is_ff & ~b_frac_is_00;
wire a_is_0   = a_expo_is_00 &  a_frac_is_00;
wire b_is_0   = b_expo_is_00 &  b_frac_is_00;
wire is_inf_nan = a_is_inf | b_is_inf | a_is_nan | b_is_nan;
wire s_is_nan = a_is_nan | (a_is_inf & b_is_0) |
                b_is_nan | (b_is_inf & a_is_0);
wire [22:0] nan_frac = ({1'b0,a[22:0]} > {1'b0,b[22:0]}) ?
                        {1'b1,a[21:0]} : {1'b1,b[21:0]};
wire [22:0] inf_nan_frac = s_is_nan? nan_frac : 23'h0;
```

以下代码生成结果符号以及临时阶码 exp10。由于计算规格化数的绝对值时要从阶码中减去 127，而在计算非规格化数的绝对值时减 126，我们在计算 exp10 时加入了 a_expo_is_00 和 b_expo_is_00。

```
wire sign = a[31] ^ b[31];
wire [9:0] exp10 = {2'h0,a[30:23]} + {2'h0,b[30:23]} - 10'h7f +
                   a_expo_is_00 + b_expo_is_00; // -126
```

以下代码使用 24 位 Wallace 树型乘法器产生部分积，然后对部分积相加。有关 Wallace 树型乘法算法的具体细节，请参阅第 3 章。

```
wire [23:0] a_frac24 = {~a_expo_is_00,a[22:0]};
wire [23:0] b_frac24 = {~b_expo_is_00,b[22:0]};
wire [47:0] z;
wire [38:0] z_sum;
wire [39:0] z_carry;
wallace_tree24 wt24 (a_frac24,b_frac24,z_sum,z_carry,z[7:0]);
assign z[47:8] = {1'b0,z_sum} + z_carry;
```

以下代码根据相加结果生成尾数并对尾数进行舍入操作。

```
wire [46:0] z5,z4,z3,z2,z1,z0;  // x.fffffffffffffffffffff...
wire [5:0] zeros;
assign zeros[5] = ~|z[46:15];                          // 32-bit 0
assign z5 = zeros[5]? {z[14:0],32'b0} : z[46:0];
assign zeros[4] = ~|z5[46:31];                         // 16-bit 0
assign z4 = zeros[4]? {z5[30:0],16'b0} : z5;
assign zeros[3] = ~|z4[46:39];                         //  8-bit 0
assign z3 = zeros[3]? {z4[38:0], 8'b0} : z4;
assign zeros[2] = ~|z3[46:43];                         //  4-bit 0
assign z2 = zeros[2]? {z3[42:0], 4'b0} : z3;
assign zeros[1] = ~|z2[46:45];                         //  2-bit 0
assign z1 = zeros[1]? {z2[44:0], 2'b0} : z2;
assign zeros[0] =  ~z1[46];                            //  1-bit 0
```

```
assign z0 = zeros[0]? {z1[45:0], 1'b0} : z1;
reg [9:0]     exp0;
reg [46:0]    frac0;
always @ * begin
    if (z[47]) begin
        exp0 = exp10 + 10'h1;     // 1x.xxxxxxxxxxxxxxxxxxxxxx xxx
        frac0 = z[47:1];
    end else begin
        if (!exp10[9] && (exp10[8:0] > zeros) && z0[46]) begin
            exp0 = exp10 - zeros; // 01.xxxxxxxxxxxxxxxxxxxxxx xxx
            frac0 = z0;
        end else begin
            exp0 = 0;             // is a denormalized number or 0
            if (!exp10[9] && (exp10 != 0))
                frac0 = z[46:0] << (exp10 - 10'h1); // e-127 --> -126
            else frac0 = z[46:0] >> (10'h1 - exp10); // e = 0 or neg
        end
    end
end
wire [26:0] frac = {frac0[46:21],|frac0[20:0]};
wire frac_plus_1 =
     ~rm[1] & ~rm[0] &  frac0[2] & (frac0[1] |  frac0[0]) |
     ~rm[1] & ~rm[0] &  frac0[2] & ~frac0[1] & ~frac0[0]  &  frac0[3] |
     ~rm[1] &  rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) &  sign |
      rm[1] & ~rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) & ~sign;
wire [24:0] frac_round = {1'b0,frac[26:3]} + frac_plus_1;
wire [9:0]  exp1 = frac_round[24]? exp0 + 10'h1 : exp0;
wire        overflow = (exp0 >= 10'h0ff) | (exp1 >= 10'h0ff);
```

最后，根据结果是否为无穷大或 NaN，选择出最后的 32 位单精度浮点结果 s。

```
wire [7:0]  final_exponent;
wire [22:0] final_fraction;
assign {final_exponent,final_fraction} = final_result(overflow, rm,
        sign, is_inf_nan, exp1[7:0], frac_round[22:0], inf_nan_frac);
assign s = {sign,final_exponent,final_fraction};
function [30:0]  final_result;
    input          overflow;
    input  [1:0]   rm;
    input          sign, is_inf_nan;
    input  [7:0]   exponent;
    input  [22:0]  fraction, inf_nan_frac;
    casex ({overflow, rm, sign, is_inf_nan})
        5'b1_00_x_x : final_result = {8'hff,23'h000000};   // inf
        5'b1_01_0_x : final_result = {8'hfe,23'h7fffff};   // max
        5'b1_01_1_x : final_result = {8'hff,23'h000000};   // inf
        5'b1_10_0_x : final_result = {8'hff,23'h000000};   // inf
        5'b1_10_1_x : final_result = {8'hfe,23'h7fffff};   // max
```

```
            5'b1_11_x_x : final_result = {8'hfe,23'h7fffff};   // max
            5'b0_xx_x_0 : final_result = {exponent,fraction};  // normal
            5'b0_xx_x_1 : final_result = {8'hff,inf_nan_frac}; // inf_nan
            default     : final_result = {8'h00,23'h000000};   // 0
        endcase
    endfunction
endmodule
```

图 9.14 给出仿真结果。其输入值与运行 C 程序时的输入值 (见表 9.4) 相同。输出结果除第 7 项之外与 C 程序的输出结果相同。第 7 项是 $(+\infty) \times (+0)$，二者的结果都是 NaN，但符号位不同。

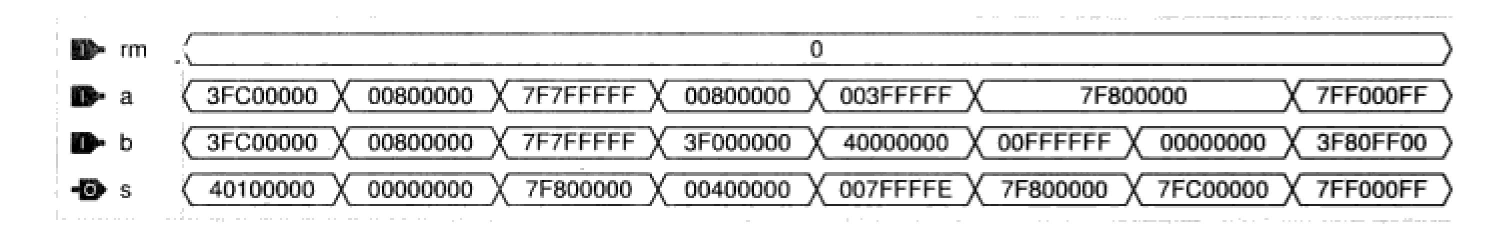


图 9.14 浮点乘法器仿真结果

由于 wallace_tree24 模块的源代码与我们在第 3 章描述的 8 位版本类似，这里就没有列出。注意它只完成部分积的计算，输出加法的和位 z_sum 及进位位 z_carry。输入 a 和 b 是两个 24 位二进制数。另外，输出 z 是 Wallace 树型乘法器的低 8 位结果。高 40 位结果由主模块 fmul 把 z_sum 和 z_carry 相加得到。图 9.15 ~ 图 9.17 合在一起是 24 位 Wallace 树型乘法器电路的结构图。

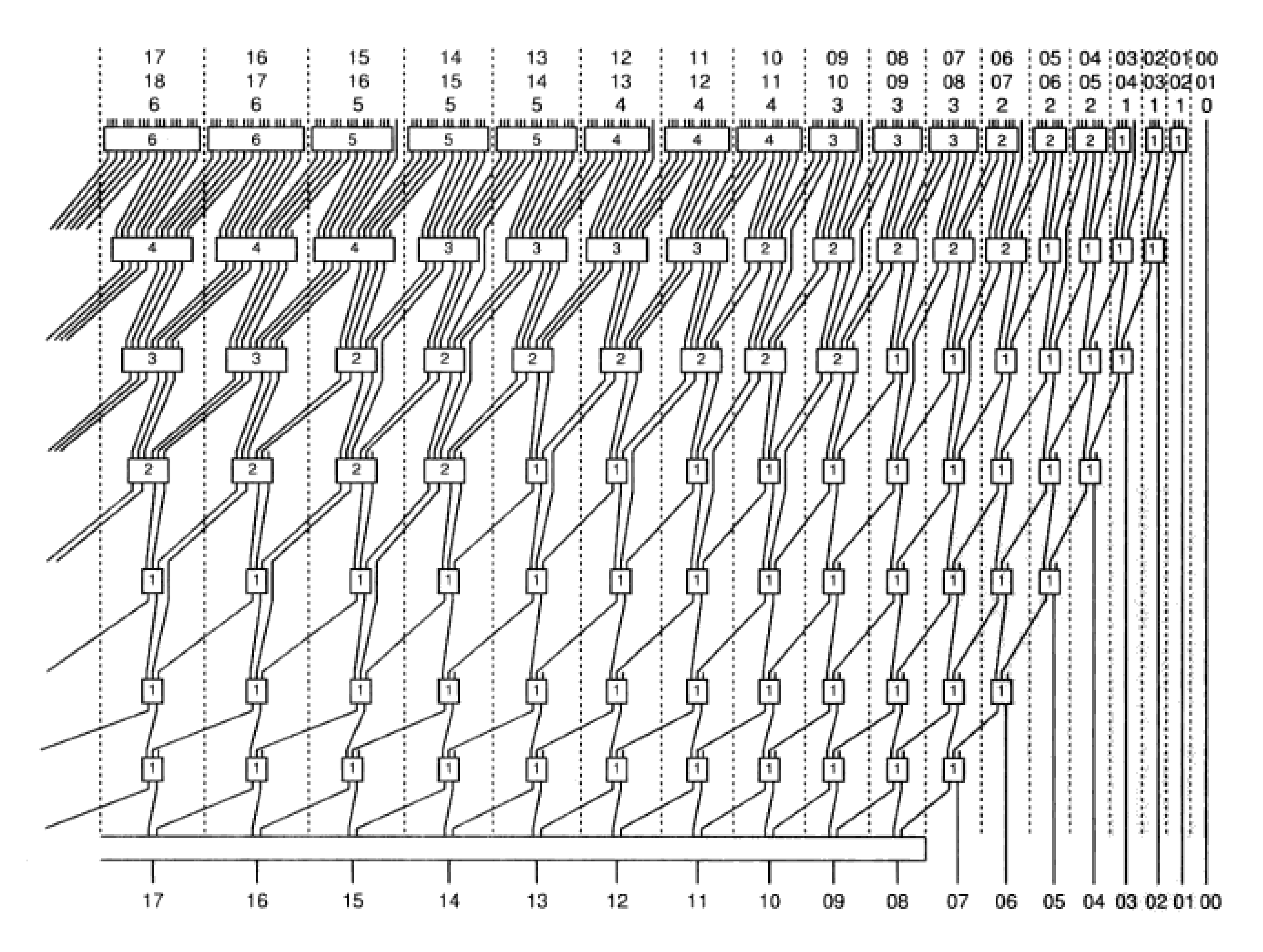


图 9.15 24 位 Wallace 树型乘法器 —— 结果 17 ~ 00 位

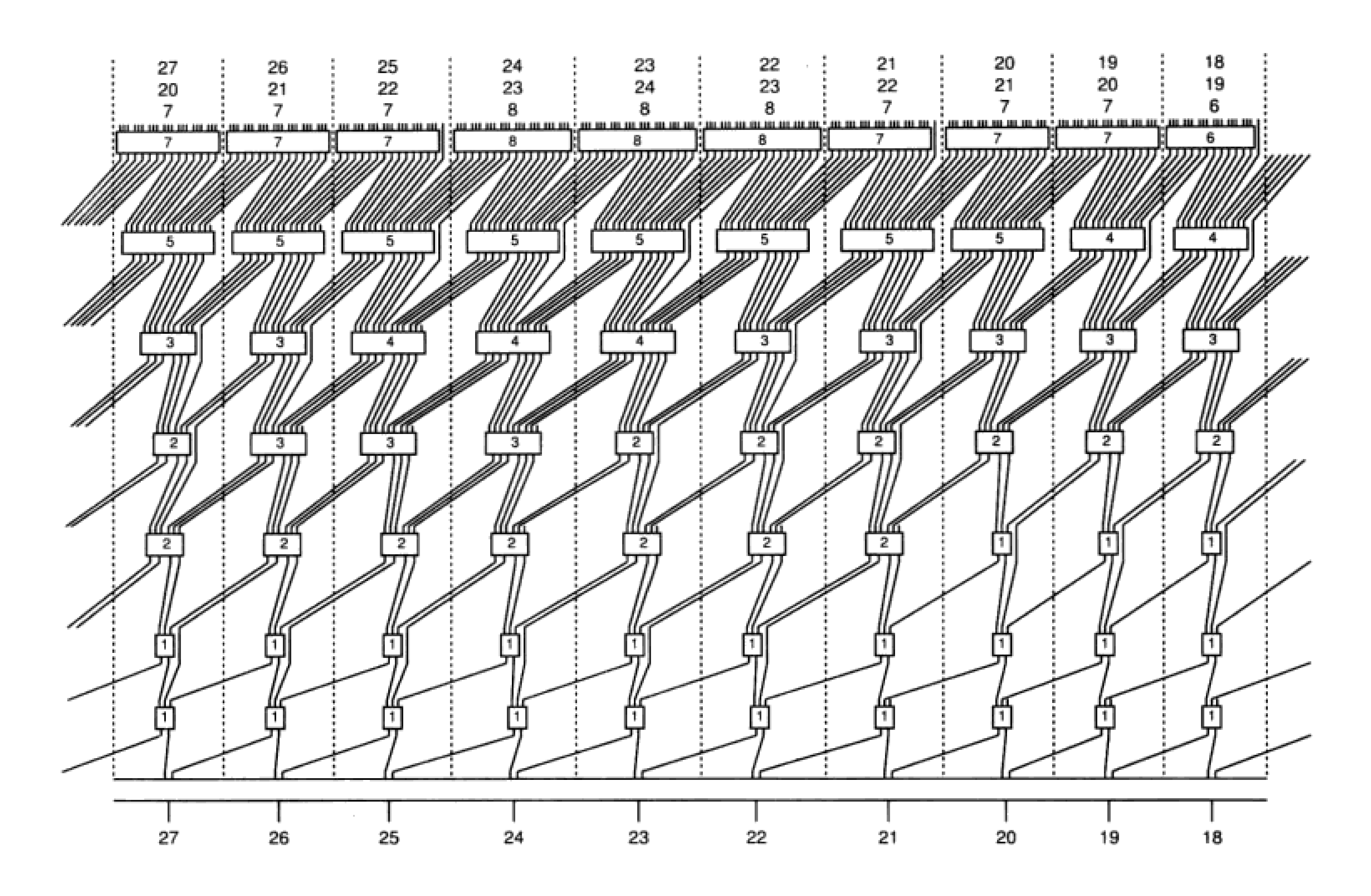


图 9.16 24 位 Wallace 树型乘法器 —— 结果 27 ~ 18 位

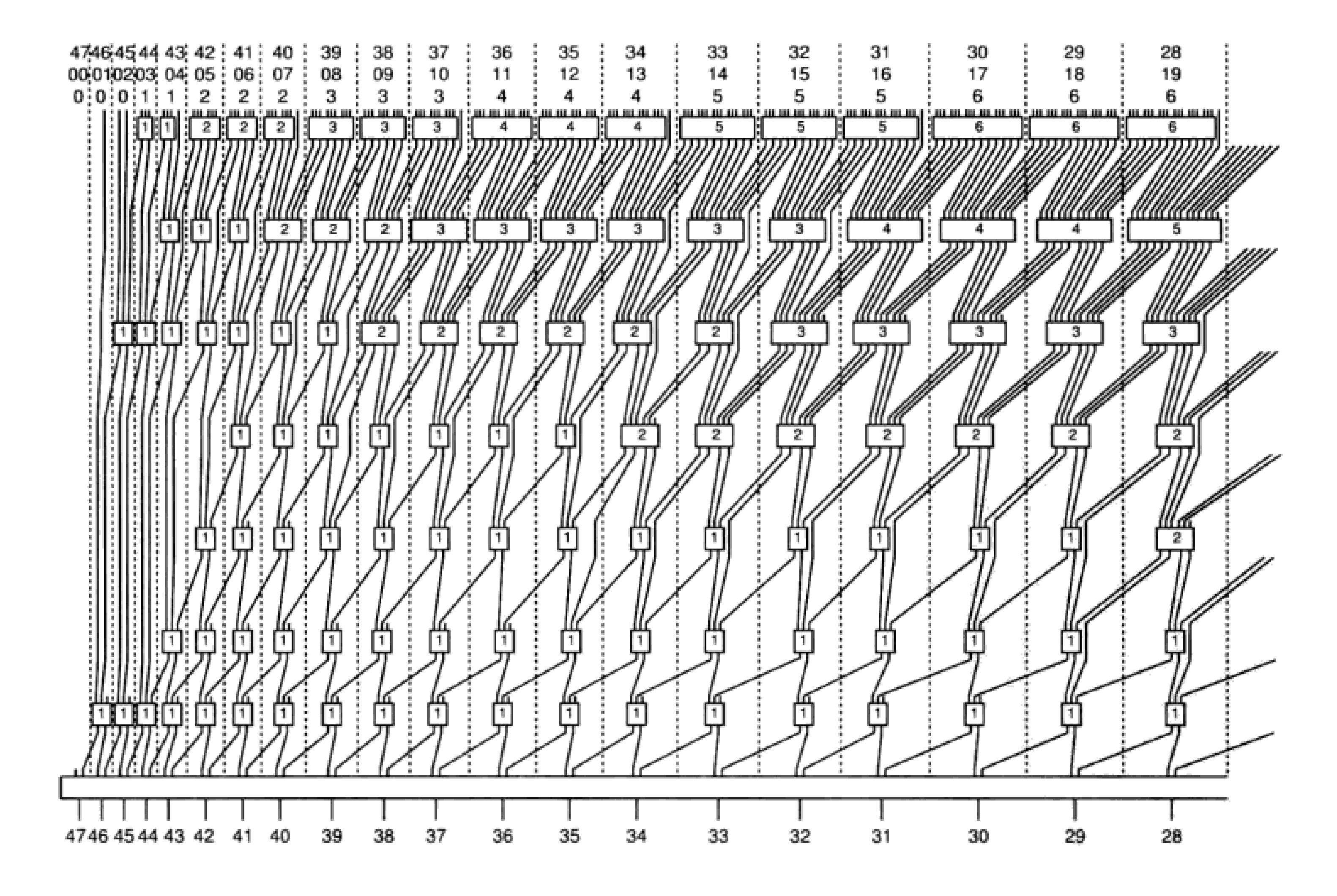


图 9.17 24 位 Wallace 树型乘法器 —— 结果 47 ~ 28 位

以上结构图仅供读者在书写 24 位 Wallace 树型乘法器的 Verilog HDL 代码时参考。图中第 1 行数字是乘积位的编号，第 2 行的数字是相应位的乘积项数量，第 3 行的数字是相应位在第 1 级所使用的全加器数量。

9.4.3 流水线 Wallace 树型浮点乘法器设计

流水线浮点乘法器能在每个周期接收一条浮点乘法指令，其电路结构如图 9.18 所示，由 3 级组成。第 1 级由 Wallace 树型乘法器完成部分积的计算，第 2 级对部分积相加，第 3 级完成规格化操作。级与级之间插入流水线寄存器。

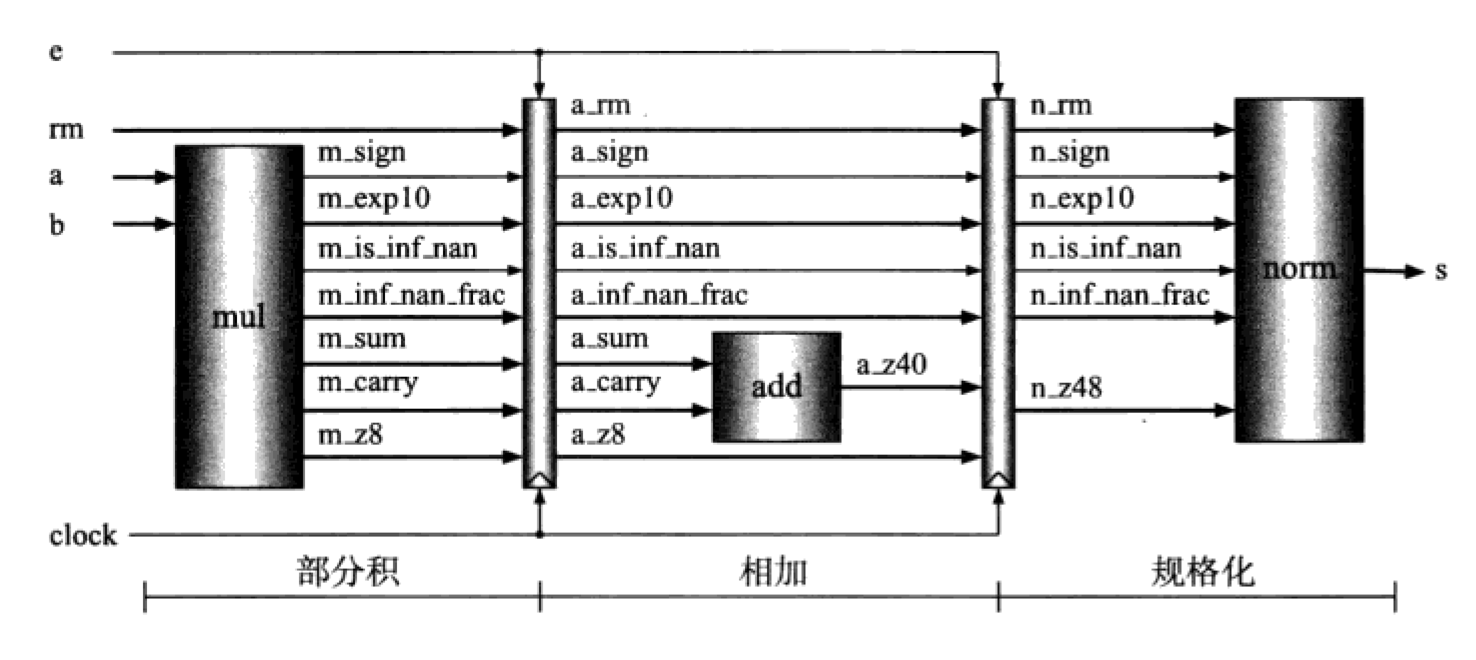


图 9.18 流水线浮点乘法器

流水线浮点乘法器的 Verilog HDL 顶层代码列在下面，它调用 5 个模块，分别是：(1) 部分积产生模块 fmul_mul；(2) 第 1 级与第 2 级之间的流水线寄存器 reg_mul_add；(3) 部分积相加模块 fmul_add；(4) 第 2 级与第 3 级之间的流水线寄存器 reg_add_norm；(5) 规格化模块 fmul_norm。

```
module pipelined_fmul (a,b,rm,s,clock,clrn,e);
    input [31:0] a,b; // fp inputs a and b
    input        e;   // enable
    input [1:0]  rm;  // round mode
    input        clock,clrn;
    output [31:0] s;  // fp output
    wire          m_sign;
    wire [9:0]    m_exp10;
    wire          m_is_inf_nan;
    wire [22:0]   m_inf_nan_frac;
    wire [38:0]   m_sum;
    wire [39:0]   m_carry;
    wire [7:0]    m_z8;
    fmul_mul mul1 (a,b,m_sign,m_exp10,m_is_inf_nan,m_inf_nan_frac,
                   m_sum,m_carry,m_z8);
    wire [1:0]    a_rm;
```

```
    wire            a_sign;
    wire [9:0]      a_exp10;
    wire            a_is_inf_nan;
    wire [22:0]     a_inf_nan_frac;
    wire [38:0]     a_sum;
    wire [39:0]     a_carry;
    wire [7:0]      a_z8;
    reg_mul_add reg_ma (rm,m_sign,m_exp10,m_is_inf_nan,
                        m_inf_nan_frac,m_sum,m_carry,m_z8,clock,
                        clrn,e, a_rm,a_sign,a_exp10,a_is_inf_nan,
                        a_inf_nan_frac,a_sum,a_carry,a_z8);
    wire [47:8]     a_z40;
    fmul_add mul2 (a_sum,a_carry,a_z40);
    wire [47:0]     a_z48 = {a_z40,a_z8};
    wire [1:0]      n_rm;
    wire            n_sign;
    wire [9:0]      n_exp10;
    wire            n_is_inf_nan;
    wire [22:0]     n_inf_nan_frac;
    wire [47:0]     n_z48;
    reg_add_norm reg_an (a_rm,a_sign,a_exp10,a_is_inf_nan,
                         a_inf_nan_frac,a_z48,clock,clrn,e,
                         n_rm,n_sign,n_exp10,n_is_inf_nan,
                         n_inf_nan_frac,n_z48);
    fmul_norm mul3 (n_rm,n_sign,n_exp10,n_is_inf_nan,
                    n_inf_nan_frac,n_z48,s);
endmodule
```

以下是部分积产生模块 fmul_mul 的代码。它的主要任务是调用 wallace_tree24 模块产生部分积，同时也处理无穷大和 NaN 等特殊情况。

```
module fmul_mul (a,b,sign,exp10,inf_nan,inf_nan_frac,
                 z_sum,z_carry,z8);
    input  [31:0] a,b;
    output        sign;
    output [9:0]  exp10;
    output        inf_nan;
    output [22:0] inf_nan_frac;
    output [38:0] z_sum;
    output [39:0] z_carry;
    output [7:0]  z8;
    wire          a_expo_is_00 = ~|a[30:23]; // exp = 00
    wire          b_expo_is_00 = ~|b[30:23];
    wire          a_expo_is_ff =  &a[30:23]; // exp = ff
    wire          b_expo_is_ff =  &b[30:23];
    wire          a_frac_is_00 = ~|a[22:0];  // frac = 0
```

```
    wire          b_frac_is_00 = ~|b[22:0];
    wire          a_is_inf = a_expo_is_ff &  a_frac_is_00;
    wire          b_is_inf = b_expo_is_ff &  b_frac_is_00;
    wire          a_is_nan = a_expo_is_ff & ~a_frac_is_00;
    wire          b_is_nan = b_expo_is_ff & ~b_frac_is_00;
    wire          a_is_0   = a_expo_is_00 &  a_frac_is_00;
    wire          b_is_0   = b_expo_is_00 &  b_frac_is_00;
    assign        inf_nan  = a_is_inf | b_is_inf |
                             a_is_nan | b_is_nan;
    wire          s_is_nan = a_is_nan | (a_is_inf &  b_is_0) |
                             b_is_nan | (b_is_inf &  a_is_0);
    wire [22:0]   nan_frac = ({1'b0,a[22:0]} > {1'b0,b[22:0]}) ?
                              {1'b1,a[21:0]} : {1'b1,b[21:0]};
    assign        inf_nan_frac = s_is_nan? nan_frac : 23'h0;
    assign        sign = a[31] ^ b[31];
    assign        exp10 = {2'h0,a[30:23]} + {2'h0,b[30:23]} -
                          10'h7f + a_expo_is_00 +
                          b_expo_is_00; // -126
    wire [23:0]   a_frac24 = {~a_expo_is_00,a[22:0]};
    wire [23:0]   b_frac24 = {~b_expo_is_00,b[22:0]};
    wallace_tree24 wt24 (a_frac24,b_frac24,z_sum,z_carry,z8);
endmodule
```

第 1 级与第 2 级之间的流水线寄存器 reg_mul_add 的代码如下。

```
module reg_mul_add (m_rm,m_sign,m_exp10,m_is_inf_nan,
    m_inf_nan_frac,m_sum,m_carry,m_z8,clock,clrn,e,a_rm,a_sign,
    a_exp10,a_is_inf_nan,a_inf_nan_frac,a_sum,a_carry,a_z8);
    input        e;    // enable
    input [1:0]  m_rm;
    input        m_sign;
    input [9:0]  m_exp10;
    input        m_is_inf_nan;
    input [22:0]  m_inf_nan_frac;
    input [38:0]  m_sum;
    input [39:0]  m_carry;
    input [7:0]   m_z8;
    input         clock,clrn;
    output [1:0]  a_rm;
    output        a_sign;
    output [9:0]  a_exp10;
    output        a_is_inf_nan;
    output [22:0]  a_inf_nan_frac;
    output [38:0]  a_sum;
    output [39:0]  a_carry;
    output [7:0]   a_z8;
```

```
    reg [1:0]     a_rm;
    reg           a_sign;
    reg [9:0]     a_exp10;
    reg           a_is_inf_nan;
    reg [22:0]    a_inf_nan_frac;
    reg [38:0]    a_sum;
    reg [39:0]    a_carry;
    reg [7:0]     a_z8;
    always @ (posedge clock or negedge clrn) begin
        if (clrn == 0) begin
            a_rm           <= 0;
            a_sign         <= 0;
            a_exp10        <= 0;
            a_is_inf_nan   <= 0;
            a_inf_nan_frac <= 0;
            a_sum          <= 0;
            a_carry        <= 0;
            a_z8           <= 0;
        end else if (e) begin
            a_rm           <= m_rm;
            a_sign         <= m_sign;
            a_exp10        <= m_exp10;
            a_is_inf_nan   <= m_is_inf_nan;
            a_inf_nan_frac <= m_inf_nan_frac;
            a_sum          <= m_sum;
            a_carry        <= m_carry;
            a_z8           <= m_z8;
        end
    end
endmodule
```

部分积相加模块 fmul_add 非常简单。

```
module fmul_add (z_sum,z_carry,z);
    input [38:0] z_sum;
    input [39:0] z_carry;
    output [47:8] z;
    assign z = {1'b0,z_sum} + z_carry;
endmodule
```

以下是第 2 级与第 3 级之间的流水线寄存器 reg_add_norm 的代码。

```
module reg_add_norm (
    a_rm,a_sign,a_exp10,a_is_inf_nan,a_inf_nan_frac,a_z48,clock,
    clrn,e,n_rm,n_sign,n_exp10,n_is_inf_nan,n_inf_nan_frac,n_z48);
    input        e;   // enable
    input [1:0] a_rm;
```

```
    input         a_sign;
    input [9:0] a_exp10;
    input         a_is_inf_nan;
    input [22:0] a_inf_nan_frac;
    input [47:0] a_z48;
    input          clock,clrn;
    output [1:0] n_rm;
    output        n_sign;
    output [9:0] n_exp10;
    output        n_is_inf_nan;
    output [22:0] n_inf_nan_frac;
    output [47:0] n_z48;
    reg [1:0]     n_rm;
    reg           n_sign;
    reg [9:0]     n_exp10;
    reg           n_is_inf_nan;
    reg [22:0]    n_inf_nan_frac;
    reg [47:0]    n_z48;
    always @ (posedge clock or negedge clrn) begin
        if (clrn == 0) begin
            n_rm            <= 0;
            n_sign          <= 0;
            n_exp10         <= 0;
            n_is_inf_nan    <= 0;
            n_inf_nan_frac <= 0;
            n_z48           <= 0;
        end else if (e) begin
            n_rm            <= a_rm;
            n_sign          <= a_sign;
            n_exp10         <= a_exp10;
            n_is_inf_nan    <= a_is_inf_nan;
            n_inf_nan_frac <= a_inf_nan_frac;
            n_z48           <= a_z48;
        end
    end
endmodule
```

规格化模块 fmul_norm 的代码如下。输出是乘积 s，32 位的单精度浮点数。

```
module fmul_norm (rm,sign,exp10,is_inf_nan,inf_nan_frac,z,s);
    input [1:0] rm;
    input         sign;
    input [9:0] exp10;
    input         is_inf_nan;
    input [22:0] inf_nan_frac;
    input [47:0] z;
    output [31:0] s;
```

```
wire [46:0]   z5,z4,z3,z2,z1,z0;  // x.fffffffffffffffffffff...
wire [5:0]    zeros;
assign        zeros[5] = ~|z[46:15];                      // 32-bit 0
assign        z5 = zeros[5]? {z[14:0],32'b0} : z[46:0];
assign        zeros[4] = ~|z5[46:31];                     // 16-bit 0
assign        z4 = zeros[4]? {z5[30:0],16'b0} : z5;
assign        zeros[3] = ~|z4[46:39];                     //  8-bit 0
assign        z3 = zeros[3]? {z4[38:0], 8'b0} : z4;
assign        zeros[2] = ~|z3[46:43];                     //  4-bit 0
assign        z2 = zeros[2]? {z3[42:0], 4'b0} : z3;
assign        zeros[1] = ~|z2[46:45];                     //  2-bit 0
assign        z1 = zeros[1]? {z2[44:0], 2'b0} : z2;
assign        zeros[0] =  ~z1[46];                        //  1-bit 0
assign        z0 = zeros[0]? {z1[45:0], 1'b0} : z1;
reg [9:0]     exp0;
reg [46:0]    frac0;
always @ * begin
    if (z[47]) begin
        exp0 = exp10 + 10'h1;     // 1x.xxxxxxxxxxxxxxxxxxxxxx xxx
        frac0 = z[47:1];
    end else begin
        if (!exp10[9] && (exp10[8:0] > zeros) && z0[46]) begin
            exp0 = exp10 - zeros; // 01.xxxxxxxxxxxxxxxxxxxxxx xxx
            frac0 = z0;
        end else begin
            exp0 = 0;             // is a denormalized number or 0
            if (!exp10[9] && (exp10 != 0))
                frac0 = z[46:0] << (exp10 - 10'h1); // e-127 --> -126
            else frac0 = z[46:0] >> (10'h1 - exp10); // e = 0 or neg
        end
    end
end
wire [26:0] frac = {frac0[46:21],|frac0[20:0]};
wire frac_plus_1 =
     ~rm[1] & ~rm[0] &  frac0[2] & (frac0[1] |  frac0[0]) |
     ~rm[1] & ~rm[0] &  frac0[2] & ~frac0[1] & ~frac0[0]  &  frac0[3] |
     ~rm[1] &  rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) &  sign |
      rm[1] & ~rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) & ~sign;
wire [24:0] frac_round = {1'b0,frac[26:3]} + frac_plus_1;
wire [9:0]  exp1 = frac_round[24]? exp0 + 10'h1 : exp0;
wire        overflow = (exp0 >= 10'h0ff) | (exp1 >= 10'h0ff);
wire [7:0]  final_exponent;
wire [22:0] final_fraction;
assign {final_exponent,final_fraction} = final_result(overflow, rm,
        sign, is_inf_nan, exp1[7:0], frac_round[22:0], inf_nan_frac);
assign s = {sign,final_exponent,final_fraction};
function [30:0]  final_result;
```

```
        input         overflow;
        input [1:0]   rm;
        input         sign, is_inf_nan;
        input [7:0]   exponent;
        input [22:0]  fraction, inf_nan_frac;
        casex ({overflow, rm, sign, is_inf_nan})
            5’b1_00_x_x : final_result = {8’hff,23’h000000};   // inf
            5’b1_01_0_x : final_result = {8’hfe,23’h7fffff};   // max
            5’b1_01_1_x : final_result = {8’hff,23’h000000};   // inf
            5’b1_10_0_x : final_result = {8’hff,23’h000000};   // inf
            5’b1_10_1_x : final_result = {8’hfe,23’h7fffff};   // max
            5’b1_11_x_x : final_result = {8’hfe,23’h7fffff};   // max
            5’b0_xx_x_0 : final_result = {exponent,fraction};  // normal
            5’b0_xx_x_1 : final_result = {8’hff,inf_nan_frac}; // inf_nan
            default     : final_result = {8’h00,23’h000000};   // 0
        endcase
    endfunction
endmodule
```

图 9.19 是流水线浮点乘法器的仿真结果。图中左上的数字 1，2 和 3 表示浮点数 3FC00000 乘以 3FC00000 的流水线级数。在第 3 级出来结果，即 40100000。

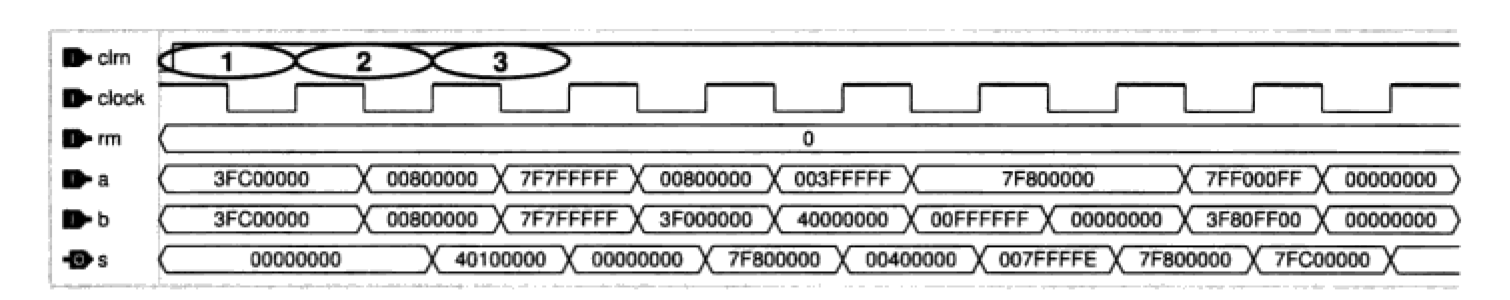


图 9.19 流水线浮点乘法器仿真结果

9.5 浮点除法器 FDIV 设计

本节首先讨论浮点除法算法，然后给出用 Newton-Raphson 算法设计的浮点除法器的 Verilog HDL 源代码。

9.5.1 浮点除法算法

浮点除法算法与浮点乘法算法基本相似，主要的不同点就是阶码的运算。首先考虑两个规格化浮点数的除法操作。设 $a = \{s_a, e_a, f_a\}$，$b = \{s_b, e_b, f_b\}$ 是两个 IEEE 754 单精度浮点数，试计算 $c = \{s_c, e_c, f_c\} = a/b$。结果 c 的符号 $s_c = s_a \oplus s_b$，c 的绝对值 $|c| = |a|/|b| = (2^{e_a-127} \times 1.f_a)/(2^{e_b-127} \times 1.f_b) = 2^{(e_a-e_b+127)-127} \times (1.f_a/1.f_b)$。我们有 $0.5 < (1.f_a/1.f_b) < 2.0$。如果 $1.f_a/1.f_b \geqslant 1.0$，则 $e_c = e_a - e_b + 127$，$1.f_c = 1.f_a/1.f_b$，否则 $e_c = e_a - e_b + 127 - 1$，$1.f_c = (1.f_a/1.f_b) << 1$ (尾数左移一位，阶码减 1)。

因为规格化数的阶码 e 满足 $1 \leqslant e \leqslant 254$，所以 $-126 \leqslant (e_a - e_b + 127) \leqslant 380$，$-127 \leqslant (e_a - e_b + 127 - 1) \leqslant 379$。即，$e_c$ 有可能超出 $1 \sim 254$ 的范围。当 $1 \leqslant e_c \leqslant 254$ 时，除法结果为规格化数；如果 $e_c > 254$，结果用无穷大 ($e_c = 255$，$f_c = 0$) 表示；当 $e_c < 1$ 时，如果除法结果大于或等于 $2^{-126} \times 0.00000000000000000000001 = 2^{-149}$，可以用非规格化数表示，否则用 0 表示。

规格化数的阶码 e 满足 $1 \leqslant e \leqslant 254$ 意味着实际的阶码 e' (即 $e - 127$) 满足 $-126 \leqslant e' \leqslant 127$。设 $a = \{s_a, e_a, f_a\}$ 为规格化数，$b = \{s_b, e_b, f_b\}$ 为非规格化数 ($e_b = 0$, $f_b \neq 0$)。$c = a/b$ 的绝对值 $|c| = |a|/|b| = (2^{e_a-127} \times 1.f_a)/(2^{-126} \times 0.f_b) = 2^{e_a-1} \times (1.f_a/0.f_b)$。最大绝对值为 $2^{254-1} \times (2 - 2^{-23})/2^{-23} = 2^{276} \times (2 - 2^{-23})$，应设结果为无穷大。最小绝对值为 $2^{1-1} \times 1.0/(1 - 2^{-23})$，这是一个规格化数。

设 $a = \{s_a, e_a, f_a\}$，$b = \{s_b, e_b, f_b\}$ 都是非规格化数。$c = a/b$ 的绝对值 $|c| = |a|/|b| = (2^{-126} \times 0.f_a)/(2^{-126} \times 0.f_b) = 0.f_a/0.f_b$。不论 f_a 和 f_b 为何值，结果都是一个规格化数。

最后我们讨论几种特殊的运算：(1) 假设 $a \neq \infty$ 并且 $a \neq \text{NaN}$，则 $a/\infty = 0$；(2) 假设 $a \neq 0$ 并且 $a \neq \text{NaN}$，则 $a/0 = \infty$；(3) $0/0 = \text{NaN}$；(4) $\infty/\infty = \text{NaN}$；(5) $\text{NaN}/b = \text{NaN}$。

9.5.2 Newton-Raphson 浮点除法器 Verilog HDL 代码

Newton-Raphson 除法算法已在第 3 章中给出，它包括两部分操作：(1) 迭代操作：$x_{i+1} = x_i(2 - x_i b)$；(2) 商的计算：$q = a \times x_n$。Newton-Raphson 浮点除法器电路的总体结构如图 9.20 所示。

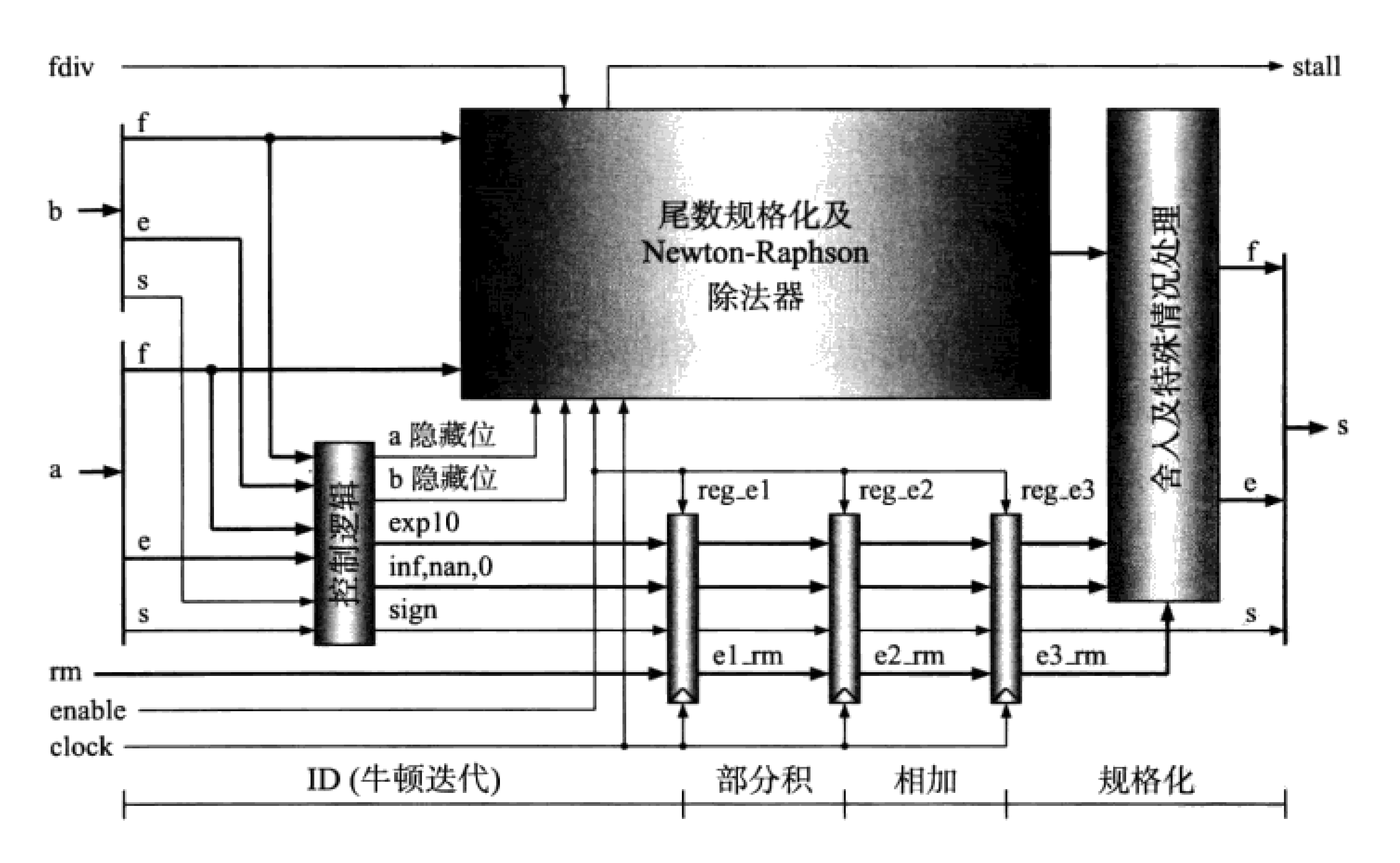


图 9.20 Newton-Raphson 浮点除法器电路的总体模块图

为了向浮点乘法流水线靠近，我们在计算 $x_{i+1} = x_i(2 - x_ib)$ 时暂停流水线，而在计算结果 $a \times x_n$ 时启动流水线。图 9.21 是 Newton-Raphson 浮点除法器的流水线示意图。查表得出 x_0 用一个周期，3 次牛顿迭代用 15 个周期。这 16 个周期由 stall 信号暂停流水线。然后是两个周期的乘法和一个周期的规格化。这部分与浮点乘法器类似，用流水线方式实现。将在下面描述如何产生 stall 信号。

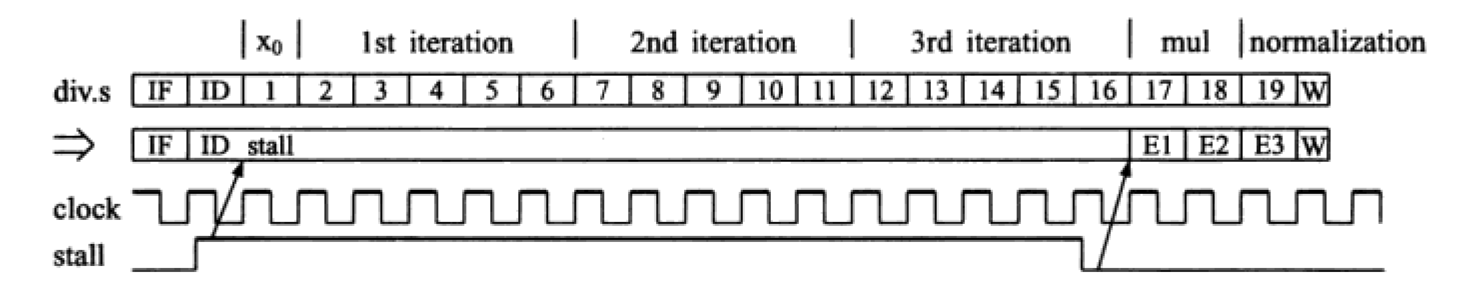


图 9.21 浮点除法指令的流水线

以下是 Newton-Raphson 浮点除法器的 Verilog HDL 代码。输出信号 count 是为演示仿真结果方便起见所设置，它只是一个内部信号。

```
module fdiv_newton (a,b,rm,fdiv,enable,clock,resetn,
                    s,busy,stall,count,reg_x);
    input [31:0] a,b;      // fp a / b
    input [1:0]  rm;       // round mode
    input        fdiv;     // ID stage: i_fdiv
    input        enable,clock,resetn; // enable
    output [31:0] s;       // fp output
    output        busy;    // for generating stall
    output        stall;   // for pipeline stall
    output [4:0] count;    // for iteration control
    output [25:00] reg_x; // x_i
    parameter ZERO = 31'h00000000;
    parameter INF  = 31'h7f800000;
    parameter NaN  = 31'h7fc00000;
    parameter MAX  = 31'h7f7fffff;
```

以下代码判断两个输入的浮点数是否为特殊的浮点数，如阶码是否为 0，是否为全 1，尾数是否为 0 等。

```
    wire  a_expo_is_00 = ~|a[30:23]; // a_expo = 00
    wire  b_expo_is_00 = ~|b[30:23]; // b_expo = 00
    wire  a_expo_is_ff =  &a[30:23]; // a_expo = ff
    wire  b_expo_is_ff =  &b[30:23]; // b_expo = ff
    wire  a_frac_is_00 = ~|a[22:0];  // a_frac = 00
    wire  b_frac_is_00 = ~|b[22:0];  // b_frac = 00
```

结果的符号由两个输入浮点数符号位“异或”得到。阶码暂时定为 $e_a - e_b + 127$，后面还要调整。非规格化数也使用上式计算阶码，但要把尾数左移一位。这时的尾数已经加入了隐藏位。

```
wire sign = a[31] ^ b[31];
wire [9:0]  exp_10 = {2'h0,a[30:23]} - {2'h0,b[30:23]} + 10'h7f;
wire [23:0] a_temp24 = a_expo_is_00? {a[22:0],1'b0} : {1'b1,a[22:0]};
wire [23:0] b_temp24 = b_expo_is_00? {b[22:0],1'b0} : {1'b1,b[22:0]};
```

由于我们使用 Newton-Raphson 算法，尾数的最高位必须为 1。这对规格化浮点数来讲没有问题，但我们的除法电路也允许非规格化浮点数参加计算，因此要把非规格化浮点的尾数调整为最高位也为 1 的格式。调整的方法是把尾数左移，直到最高位为 1 为止。这部分电路由另一个模块实现，模块名为 shift_to_msb_equ_1，将在下面给出。移位时要记录浮点数 a 和 b 各自左移了多少位，然后用它们来调整阶码，以保证浮点数值大小不变。

```
wire [23:0] a_frac24,b_frac24; // to 1xx...x for den
wire [4:0]  shamt_a,shamt_b;   // how many bits shifted
shift_to_msb_equ_1 shift_a (a_temp24,a_frac24,shamt_a);
shift_to_msb_equ_1 shift_b (b_temp24,b_frac24,shamt_b);
wire [9:0]     exp10 = exp_10 - shamt_a + shamt_b;
```

以下代码实现图 9.20 中的三个流水线寄存器 reg_e1，reg_e2 和 reg_e3。

```
reg e1_sign,e1_ae00,e1_aeff,e1_af00,e1_be00,e1_beff,e1_bf00;
reg e2_sign,e2_ae00,e2_aeff,e2_af00,e2_be00,e2_beff,e2_bf00;
reg e3_sign,e3_ae00,e3_aeff,e3_af00,e3_be00,e3_beff,e3_bf00;
reg [1:0] e1_rm,e2_rm,e3_rm;
reg [9:0] e1_exp10,e2_exp10,e3_exp10;
always @ (negedge resetn or posedge clock)
  if (resetn == 0) begin // pipeline registers
      // reg_e1                // reg_e2                // reg_e3
      e1_sign  <= 0;           e2_sign  <= 0;           e3_sign  <= 0;
      e1_rm    <= 0;           e2_rm    <= 0;           e3_rm    <= 0;
      e1_exp10 <= 0;           e2_exp10 <= 0;           e3_exp10 <= 0;
      e1_ae00  <= 0;           e2_ae00  <= 0;           e3_ae00  <= 0;
      e1_aeff  <= 0;           e2_aeff  <= 0;           e3_aeff  <= 0;
      e1_af00  <= 0;           e2_af00  <= 0;           e3_af00  <= 0;
      e1_be00  <= 0;           e2_be00  <= 0;           e3_be00  <= 0;
      e1_beff  <= 0;           e2_beff  <= 0;           e3_beff  <= 0;
      e1_bf00  <= 0;           e2_bf00  <= 0;           e3_bf00  <= 0;
  end else if (enable) begin
      e1_sign  <= sign;     e2_sign  <= e1_sign;  e3_sign  <= e2_sign;
      e1_rm    <= rm;       e2_rm    <= e1_rm;    e3_rm    <= e2_rm;
      e1_exp10 <= exp10;    e2_exp10 <= e1_exp10; e3_exp10 <= e2_exp10;
      e1_ae00  <= a_expo_is_00; e2_ae00 <= e1_ae00; e3_ae00 <= e2_ae00;
      e1_aeff  <= a_expo_is_ff; e2_aeff <= e1_aeff; e3_aeff <= e2_aeff;
      e1_af00  <= a_frac_is_00; e2_af00 <= e1_af00; e3_af00 <= e2_af00;
      e1_be00  <= b_expo_is_00; e2_be00 <= e1_be00; e3_be00 <= e2_be00;
      e1_beff  <= b_expo_is_ff; e2_beff <= e1_beff; e3_beff <= e2_beff;
      e1_bf00  <= b_frac_is_00; e2_bf00 <= e1_bf00; e3_bf00 <= e2_bf00;
  end
```

有了规格化的尾数，我们可以调用 Newton-Raphson 24 位除法模块了。返回的结果为尾数相除的商，用 31 位表示。如果浮点数 a 的尾数大于或等于 b 的尾数，则商有 1.xxxxx···x 的格式；否则为 0.1xxxx···x。我们把它们统一成 1.xxxxx···x 的格式。当然，如果原来是 0.1xxxx···x，要从结果的阶码中减 1。

```
newton24 frac_newton (a_frac24,b_frac24,fdiv,enable,clock,resetn,
                      q,busy,count,reg_x,stall);
wire [31:00] q; // af24/bf24 = 1.xxxxx...x or 0.1xxxx...x
wire [31:0]  z0 = q[31] ? q : {q[30:0],1'b0}; // 1.xxxxx...x
wire [9:0]   exp_adj = q[31] ? e3_exp10 : e3_exp10 - 10'b1; // reg_e3
```

以下代码对阶码和尾数做进一步的调整，这是因为结果可能为非规格化数或者为无穷大。注意这部分代码虽然使用了 always 语句，但由于在所有的条件下，两个变量均有赋值，生成的电路还是组合电路。

```
reg [9:0]    exp0;
reg [31:0]   frac0;
always @ * begin
    if (exp_adj[9]) begin // exp is negative
        exp0 = 0;
        if (z0[31]) // 1.xx...x exp_adj = minus
            frac0 = z0 >> (10'b1 - exp_adj); // den (-126)
        else frac0 = 0;
    end else if (exp_adj == 0) begin // exp is 0
        exp0 = 0;
        frac0 = {1'b0,z0[31:2],|z0[1:0]}; // den (-126)
    end else begin // exp > 0
        if (exp_adj > 254) begin // inf
            exp0 = 10'hff;
            frac0 = 0;
        end else begin // normal
            exp0 = exp_adj;
            frac0 = z0;
        end
    end
end
```

现在根据舍入方式 (rm) 对尾数进行舍入。首先把 32 位尾数变成 26 位，最低 3 位为 GRS。舍入方法与浮点加法相同。

```
wire [26:0] frac = {frac0[31:6],|frac0[5:0]}; // sticky
wire frac_plus_1 = // reg_e3
     ~e3_rm[1] & ~e3_rm[0] & frac[3] & frac[2] & ~frac[1] & ~frac[0] |
     ~e3_rm[1] & ~e3_rm[0] & frac[2] & (frac[1] | frac[0]) |
     ~e3_rm[1] &  e3_rm[0] & (frac[2] | frac[1] | frac[0]) & e3_sign |
      e3_rm[1] & ~e3_rm[0] & (frac[2] | frac[1] | frac[0]) & ~e3_sign;
wire [24:0] frac_round = {1'b0,frac[26:3]} + frac_plus_1;
```

```
wire [9:0]  exp1 = frac_round[24]? exp0 + 10'h1 : exp0;
wire        overflow = (exp1 >= 10'h0ff); // overflow
```

调用 function 对特殊浮点数进行处理。

```
wire [7:0]  exponent;
wire [22:0] fraction;
assign {exponent,fraction}=final_result(overflow,e3_rm,e3_sign,e3_ae00,
 e3_aeff,e3_af00,e3_be00,e3_beff,e3_bf00,{exp1[7:0],frac_round[22:0]});
assign s = {e3_sign,exponent,fraction};
```

下面的 function 处理上溢和特殊浮点数。对特殊浮点数的处理，浮点除法比起浮点加法和浮点乘法要复杂得多，好在我们有 casex 可用 (casex 算是一个好东西)。

```
function [30:0] final_result;
    input overflow;
    input [1:0] e3_rm;
    input e3_sign;
    input a_e00,a_eff,a_f00, b_e00,b_eff,b_f00;
    input [30:0] calc;
    casex ({overflow,e3_rm,e3_sign,a_e00,a_eff,a_f00,b_e00,b_eff,b_f00})
        10'b100x_xxx_xxx : final_result = INF;  // overflow
        10'b1010_xxx_xxx : final_result = MAX;  // overflow
        10'b1011_xxx_xxx : final_result = INF;  // overflow
        10'b1100_xxx_xxx : final_result = INF;  // overflow
        10'b1101_xxx_xxx : final_result = MAX;  // overflow
        10'b111x_xxx_xxx : final_result = MAX;  // overflow
        10'b0xxx_010_xxx : final_result = NaN;  // NaN / any
        10'b0xxx_011_010 : final_result = NaN;  // inf / NaN
        10'b0xxx_100_010 : final_result = NaN;  // den / NaN
        10'b0xxx_101_010 : final_result = NaN;  //   0 / NaN
        10'b0xxx_00x_010 : final_result = NaN;  // normal / NaN
        10'b0xxx_011_011 : final_result = NaN;  // inf / inf
        10'b0xxx_100_011 : final_result = ZERO; // den / inf
        10'b0xxx_101_011 : final_result = ZERO; //   0 / inf
        10'b0xxx_00x_011 : final_result = ZERO; // normal / inf
        10'b0xxx_011_101 : final_result = INF;  // inf / 0
        10'b0xxx_100_101 : final_result = INF;  // den / 0
        10'b0xxx_101_101 : final_result = NaN;  //   0 / 0
        10'b0xxx_00x_101 : final_result = INF;  // normal / 0
        10'b0xxx_011_100 : final_result = INF;  // inf / den
        10'b0xxx_100_100 : final_result = calc; // den / den
        10'b0xxx_101_100 : final_result = ZERO; //   0 / den
        10'b0xxx_00x_100 : final_result = calc; // normal / den
        10'b0xxx_011_00x : final_result = INF;  // inf / normal
        10'b0xxx_100_00x : final_result = calc; // den / normal
        10'b0xxx_101_00x : final_result = ZERO; //   0 / normal
        10'b0xxx_00x_00x : final_result = calc; // normal / normal
        default          : final_result = NaN;
```

```
        endcase
    endfunction
endmodule
```

以下模块把尾数左移，使其最高位为 1，同时记录左移位数。

```
module shift_to_msb_equ_1 (a,b,shamt);
    input  [23:00] a; // shift a=xx...x to b=1x...x
    output [23:00] b; // 1x...x
    output [4:0] shamt; // how many bits shifted
    wire [23:0]  a5,a4,a3,a2,a1,a0;
    assign a5 = a;
    assign shamt[4]  = ~|a5[23:08];                    // 16-bit 0
    assign a4 = shamt[4]? {a5[07:00],16'b0} : a5;
    assign shamt[3]  = ~|a4[23:16];                    //  8-bit 0
    assign a3 = shamt[3]? {a4[15:00], 8'b0} : a4;
    assign shamt[2]  = ~|a3[23:20];                    //  4-bit 0
    assign a2 = shamt[2]? {a3[19:00], 4'b0} : a3;
    assign shamt[1]  = ~|a2[23:22];                    //  2-bit 0
    assign a1 = shamt[1]? {a2[21:00], 2'b0} : a2;
    assign shamt[0]  =  ~a1[23];                       //  1-bit 0
    assign a0 = shamt[0]? {a1[22:00], 1'b0} : a1;
    assign b = a0;
endmodule
```

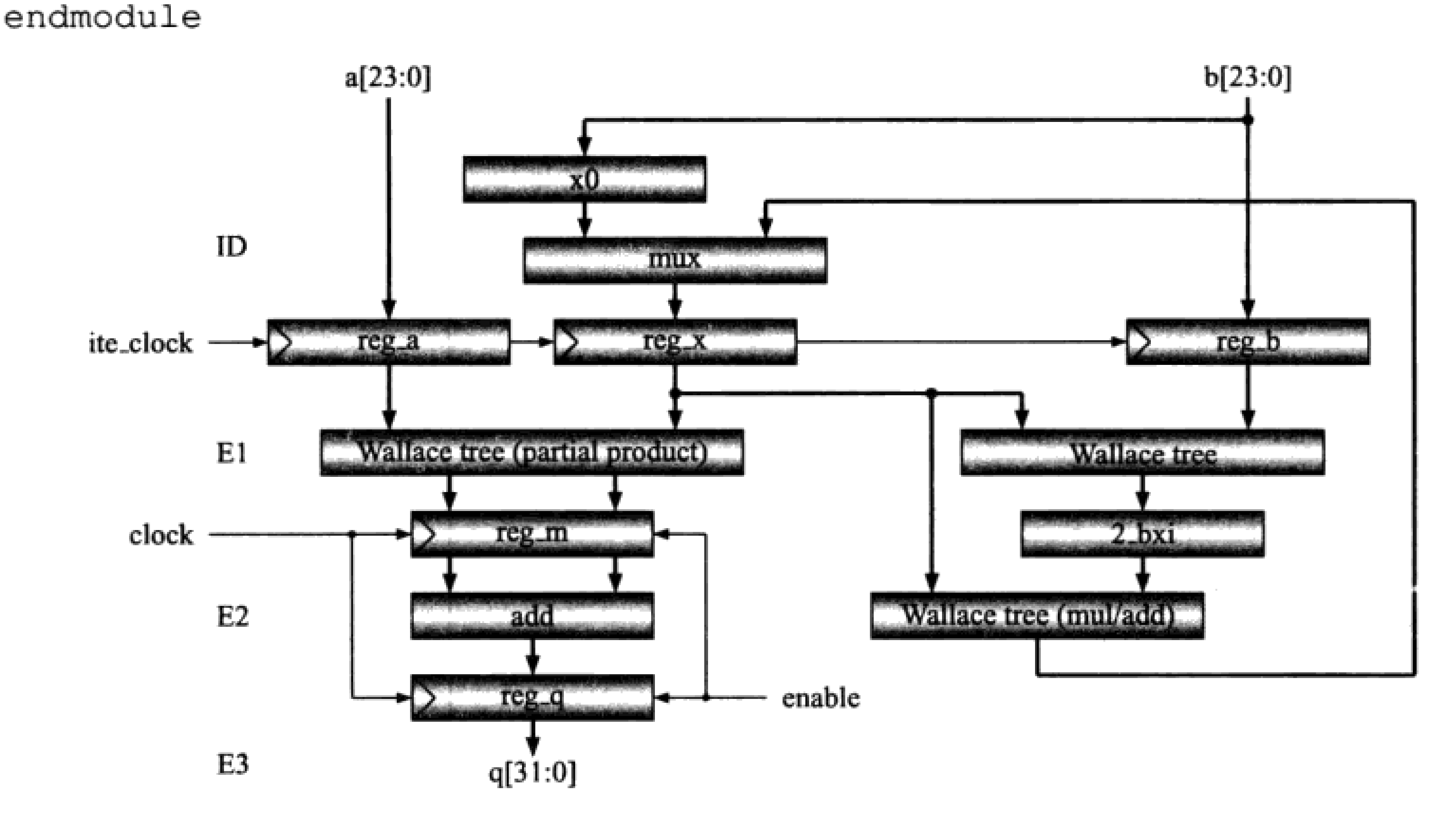


图 9.22 Newton-Raphson 除法器

以下模块完成 24 位 Newton-Raphson 除法，它与 32 位的版本类似，不同之处是最后的乘法用流水线方式实现 (见图 9.22) 以及使用了系统时钟信号。图 9.22 中的 reg_m 和 reg_q 是两级流水线寄存器。迭代完成后，立即进行部分积的运算并把结果打入 reg_m 寄存器。

```
module newton24 (a,b,fdiv,enable,clock,resetn,q,busy,count,reg_x,
                 stall);
    input [23:00] a;      // dividend: fraction: .1xxx...x
    input [23:00] b;      // divisor:  fraction: .1xxx...x
    input         fdiv;   // ID stage: i_fdiv
    input         enable,clock,resetn;
    output [31:00] q;     // a/b: x.xxxxx...x
    output        busy;   // cannot receive new div
    output [4:0]  count;  // for sim test only
    output [25:00] reg_x; // for sim test only 01.xx...x
    output        stall;  // for pipeline stall
    reg [31:00]   q;      // a/b: x.xxxxx...x
    reg [25:00]   reg_x;  // 26-bit: xx.xxxxx...xx
    reg [23:00]   reg_a;  // 24-bit:   .1xxxx...xx
    reg [23:00]   reg_b;  // 24-bit:   .1xxxx...xx
    reg [4:0]     count;  // 3 iterations
    reg           busy;
```

初始值 x_0 通过查 ROM 表得到，用一个时钟周期。由于除数的最高位为 1，访问 ROM 时没有必要使用它，但计算表中的内容时不能忘了它。当 start 信号有效时，我们对寄存器赋初值。与我们在第 3 章介绍的 32 位版本不同，我们这里使用了实际的时钟周期进行计数，而不是对迭代次数进行计数。计数器值为 0 时表示指令处在 ID 级。如果当前指令是浮点除法且计数器值为 0，把计数器设为 1，并令 busy 信号也为 1。在下一个时钟上升沿处，把数据 x_0，a 和 b 分别打入寄存器 reg_x，reg_a 和 reg_b。Newton-Raphson 的迭代公式为 $x_{i+1} = x_i(2 - x_i b)$。一次迭代用 5 个时钟周期：两次乘法用 4 个周期，一次减法用一个周期。在计算过程中，当计数值为 6，11 和 16 时，我们修改寄存器 x_i 的内容 (3 次迭代)。当计数器为 15 时，清除 busy。当计数器为 16 时，清零，以允许下一条指令的执行。当 x_3 计算出之后，下一个周期使用 Wallace 树型乘法算法计算出部分积的和 m_s 与进位 m_c，并把它们保存在 reg_m 寄存器。

```
    wire [7:0]     x0 = rom(b[22:19]);
    always @ (posedge clock or negedge resetn) begin
        if (resetn == 0) begin
            count   <= 5'b0; // reset count
            busy    <= 1'b0; // reset to not busy
        end else begin       // not reset
            if (fdiv & (count == 0)) begin // do once only
                count <= 5'b1;             // set count
                busy  <= 1'b1;             // set to busy
            end else begin  // execution: 3 iterations
                if  (count == 5'h01) begin
                    reg_x <= {2'b1,x0,16'b0}; // 01.xxxx0...0
                    reg_a <= a;               //   .1xxxx...x
```

```
                    reg_b <= b;                 //    .1xxxx...x
                end
                if  (count != 0) count <= count + 5'b1; // count++
                if  (count == 5'h0f) busy <= 0;  // ready for next
                if  (count == 5'h10) count <= 5'b0; // reset count
                if ((count == 5'h06) ||
                    (count == 5'h0b) ||
                    (count == 5'h10))
                    reg_x <= x52[50:25];       // xx.xxxxx...x
            end
        end
    end
```

我们使用简单的暂停流水线的方法等待浮点除法指令的执行。下面的 stall 信号为 1 时，流水线暂停。我们希望它的波形如图 9.23 所示，因此我们产生暂停流水线的信号 stall 如下：

```
    assign    stall = fdiv & (count == 0) | busy;
```

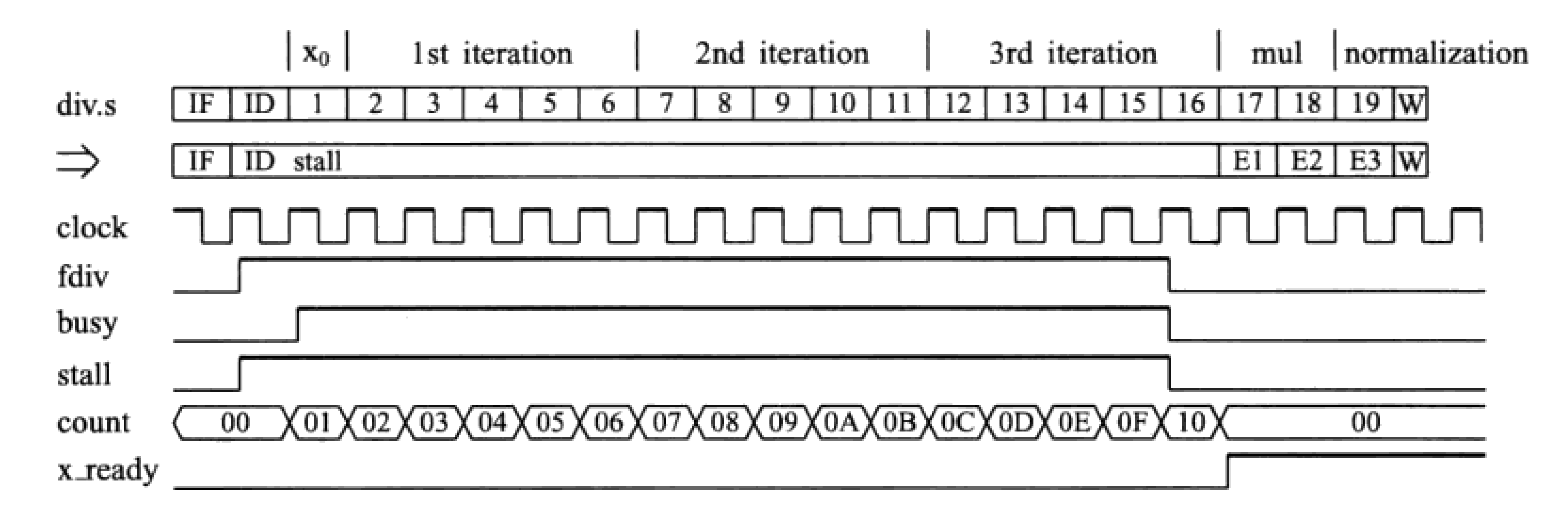


图 9.23 浮点除法指令暂停流水线的信号 stall 的产生

以下代码实现 $x_{i+1} = x_i(2 - x_i b)$。注意该处的减法实际上是求补码的运算，我们这里用了取反加 1 来实现减法。乘法用 Wallace 树型算法实现。代码太长，不再列出。读者可以参照第 3 章的 8 位 Wallace 树型乘法算法源代码和本章 24 位 Wallace 树型乘法器结构 (见图 9.15 ~ 图 9.17)，书写 wallace_tree24x26 模块。

```
    wire [49:00] bxi;                            //  xx.xxxxx...x
    wire [51:00] x52;                            // xxx.xxxxx...x
    wallace_tree26x24 bxxi (reg_x,reg_b,bxi); // bxi=reg_x*reg_b
    wire [25:00] b26 = ~bxi[48:23] + 1'b1;      //    x.xxxxx...x
    wallace_tree26x26 xip1 (reg_x,b26,x52);    // x52=reg_x*b26
```

如果 3 次迭代完成，我们用一次乘法 $(a \times x_3)$ 计算出 32 位的商。乘法用 Wallace 树型算法及流水线方式实现，即在部分积和加法器之间加了一个流水线寄存器。两

级流水线寄存器 reg_m 和 reg_q 分别存放部分积 (m_s 和 m_c) 和相加的结果 (e2p)。我们这里没有对商进行舍入，因为舍入在规格化阶段进行。

```
    wire [48:0]  m_s; // 41 + 8 = 49-bit
    wire [41:0]  m_c; // 42-bit
    wallace24x26_mul wt (reg_a,reg_x,m_s[48:8],m_c,m_s[7:0]);
    reg [48:0]   a_s; // 41-bit
    reg [41:0]   a_c; // 42-bit
    always @ (negedge resetn or posedge clock)
      if (resetn == 0) begin // registers: reg_m and reg_q
          a_s <= 0; // reg_m
          a_c <= 0; // reg_m
          q   <= 0; // reg_q
      end else if (enable) begin
          a_s <= m_s; // save partial product sum
          a_c <= m_c; // save partial product carry
          q   <= e2p;
      end
    wire [49:00] d_x = {1'b0,a_s} + {a_c,8'b0}; // 0x.xxxxx...x
    wire [31:00] e2p = {d_x[48:18],|d_x[17:0]};   // sticky
```

以下是 ROM 表，存放 x_0。最高位的 1 没有存放在表中，在赋初值时别忘了加上。虽然我们给出了 8 位初值，加上省略的 1 共 9 位，但地址位只用了 4 位，实际上这是不够的，或者说 9 位结果是不精确的。

```
    function [7:0] rom;
        input [3:0] b;
        case (b)
            4'h0: rom = 8'hf0;              4'h1: rom = 8'hd4;
            4'h2: rom = 8'hba;              4'h3: rom = 8'ha4;
            4'h4: rom = 8'h8f;              4'h5: rom = 8'h7d;
            4'h6: rom = 8'h6c;              4'h7: rom = 8'h5c;
            4'h8: rom = 8'h4e;              4'h9: rom = 8'h41;
            4'ha: rom = 8'h35;              4'hb: rom = 8'h29;
            4'hc: rom = 8'h1f;              4'hd: rom = 8'h15;
            4'he: rom = 8'h0c;              4'hf: rom = 8'h04;
        endcase
    endfunction
endmodule
```

浮点除法总共需要 19 个周期：查表用一个周期，3 次迭代 15 个，最后的乘法两个，规格化一个。但浮点除法指令的发出只需隔 16 个周期，因为最后的三个周期使用流水线方式。图 9.24 是仿真结果，第一部分是 4.0/2.0 = 2.0。第二部分是非规格化数除以非规格化数，结果为规格化数。还有很多其他的情况需要仿真，我们不在这里一一给出。读者可以写一个 C 语言程序测试 x86 浮点除法的结果，尤其是特殊数据的除法结果，并与仿真结果进行比较。

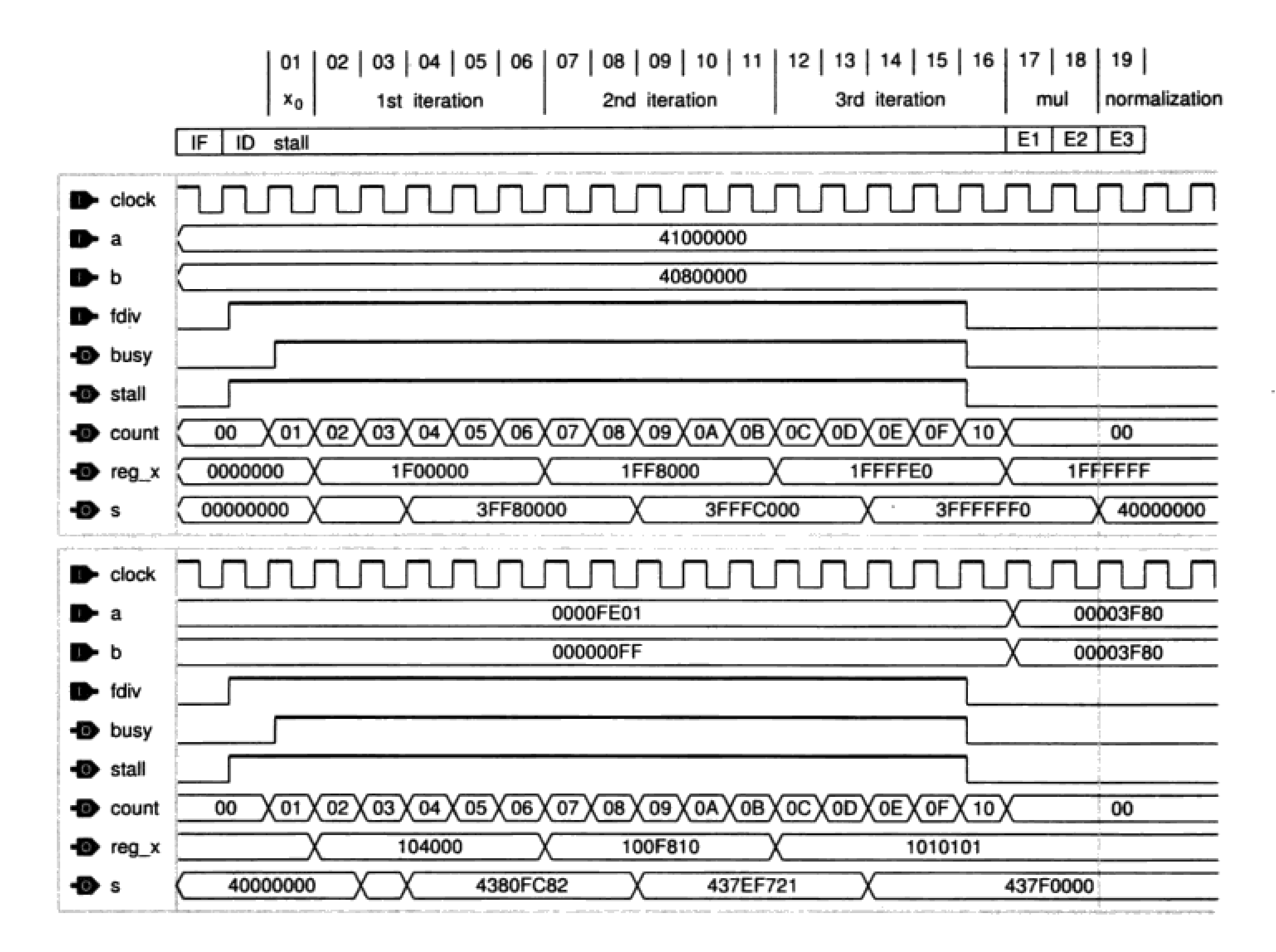


图 9.24 Newton-Raphson 浮点除法器仿真结果

9.6 浮点开方器 FSQRT 设计

本节首先讨论浮点开方算法，然后给出用 Newton-Raphson 算法设计的浮点开方电路的 Verilog HDL 源代码。

9.6.1 浮点开方算法

浮点开方算法最简单，因为它只有一个操作数并且结果朝 1.0 的方向走。设 $d = \{s_d, e_d, f_d\}$ 为 IEEE 754 单精度浮点数，试计算 $q = \{s_q, e_q, f_q\} = \sqrt{d}$。只考虑 d 为正数的情况，为负时结果是 NaN。

首先假设 d 是一个规格化数，结果应为 $\sqrt{2^{e_d-127} \times 1.f_d}$。我们采用 Newton-Raphson 算法实现浮点开方电路，因此必须把 $1.f_d$ 变成 $0.1xx\cdots x$ 或 $0.01x\cdots x$ 的格式，即把 $1.f_d$ 右移一位或两位。开方后的格式为 $0.1xx\cdots x$，再将其左移一位，变为 $1.f_q$ 的标准格式，阶码同时减 1。我们分两种情况考查 $\sqrt{2^{e_d-127} \times 1.f_d}$：

1) 如果 $e_d - 127$ 为偶数，即 e_d 为奇数，右移 $1.f_d$ 两位，$1.f_q = \sqrt{1.f_d>>2} << 1$，$e_q = (e_d - 127 + 2)/2 + 127 - 1 = e_d>>1 + 63 + e_d \% 2$；
2) 如果 $e_d - 127$ 为奇数，即 e_d 为偶数，右移 $1.f_d$ 一位，$1.f_q = \sqrt{1.f_d>>1} << 1$，$e_q = (e_d - 127 + 1)/2 + 127 - 1 = e_d>>1 + 63 + e_d \% 2$。

虽然以上两种情况 e_d 的奇偶制度不同，但我们用公式 $e_q = e_d >> 1 + 63 + e_d \% 2$ 实现了 e_q 的和平统一，方法是右移出去的 e_d 的最低位别丢，加回去。以下给出一个求规格化数平方根的例子。假设我们有一个用十六进制表示的单精度浮点数 $d = 41100000_{16}$，试计算它的平方根。把 41100000_{16} 写成二进制格式格式：

$$41100000_{16} = 0_10000010_00100000000000000000000_2,$$

即 $s_d = 0$，$e_d = 10000010_2 = 130_{10}$，$f_d = 00100000000000000000000_2$。它的十进制数值为 $2^{130-127} \times 1.00100000000000000000000_2 = 2^3 \times 1.001_2 = 1001_2 = 9_{10}$。我们知道它的平方根是 $q = 3_{10} = 11_2 = 1.1_2 \times 2^1 = 1.1_2 \times 2^{128-127}$，即 $s_q = 0$，$e_q = 10000000_2$，$f_q = 10000000000000000000000_2$，用十六进制表示为 40400000_{16}。

按照前面给出的算法，我们知道 $e_d = 10000010_2 = 130_{10}$ 是偶数，因此把 $1.f_d$ 右移一位，变为 0.1001_2。然后用 Newton-Raphson 算法求它的平方根，得到 0.11_2，再左移一位，变为 $1.1_2 = 1.f_q$。$e_q = e_d >> 1 + 63 + e_d \% 2 = 65 + 63 + 0 = 128$。我们得到了平方根 $q = 40400000_{16}$。

如果 d 是一个 IEEE 754 非规格化数，为了求出 e_q 和 f_q，我们首先把 $0.f_d$ 左移偶数位，使其有 $0.1xx\cdots x$ 或 $0.01x\cdots x$ 的格式。假设移位位数为 b (偶数)。对移位后的数据开方，结果为 $0.1xx\cdots x$，再将其左移一位，变为 $1.f_q$ 的标准格式。结果的阶码 $e_q = (-126 - b)/2 - 1 + 127 = 63 - b/2$。

以下再给出一个求非规格化数平方根的例子。假设我们有一个用十六进制表示的单精度浮点数 $d = 00003200_{16}$，试计算它的平方根。把十六进制数 00003200_{16} 写成二进制格式，我们有

$$00003200_{16} = 0_00000000_00000000011001000000000_2,$$

即 $s_d = 0$，$e_d = 00000000_2 = 0_{10}$，$f_d = 00000000011001000000000_2$。它的十进制数值为 $2^{-126} \times 0.00000000011001000000000_2 = 2^{-126} \times 11001_2 \times 2^{-14} = 2^{-140} \times 25_{10}$。我们知道它的平方根是 $q = 2^{-70} \times 5_{10} = 2^{-70} \times 101_2 = 2^{-68} \times 1.01_2 = 2^{59-127} \times 1.01_2$，即 $s_q = 0$，$e_q = 00111011_2$，$f_q = 01000000000000000000000_2$，用十六进制表示为 $1DA00000_{16}$。

按照前面给出的算法，把 $0.f_d = 0.00000000011001000000000_2$ 左移 8 位 (偶数)，变为 $0.01100100000000000000000_2$。然后用 Newton-Raphson 算法求它的平方根，得到 0.101_2，再左移一位，变为 $1.01_2 = 1.f_q$。$e_q = 63 - 8/2 = 59$。我们得到了平方根 $q = 1DA00000_{16}$。

规格化数和非规格化数的平方根均为规格化数，因此在规格化阶段只完成舍入操作。最后讨论几种特殊的运算：(1) 如果 $s_d = 1$，即 d 为负，则开方结果为 NaN；(2) $\sqrt{+\infty} = +\infty$；(3) $\sqrt{\text{NaN}} = \text{NaN}$。

9.6.2 Newton-Raphson 浮点开方器 Verilog HDL 代码

Newton-Raphson 开方算法已在第 3 章中给出，它包括两部分操作：(1) 迭代操作：$x_{i+1} = x_i(3 - x_i^2 d)/2$；(2) 平方根的计算：$q = d \times x_n$。

Newton-Raphson 浮点开方电路的总体结构如图 9.25 所示。为了向浮点乘法流水线靠近，我们在计算 $x_{i+1} = x_i(3 - x_i^2 d)/2$ 时暂停流水线，而在计算结果 $d \times x_n$ 时启动流水线。

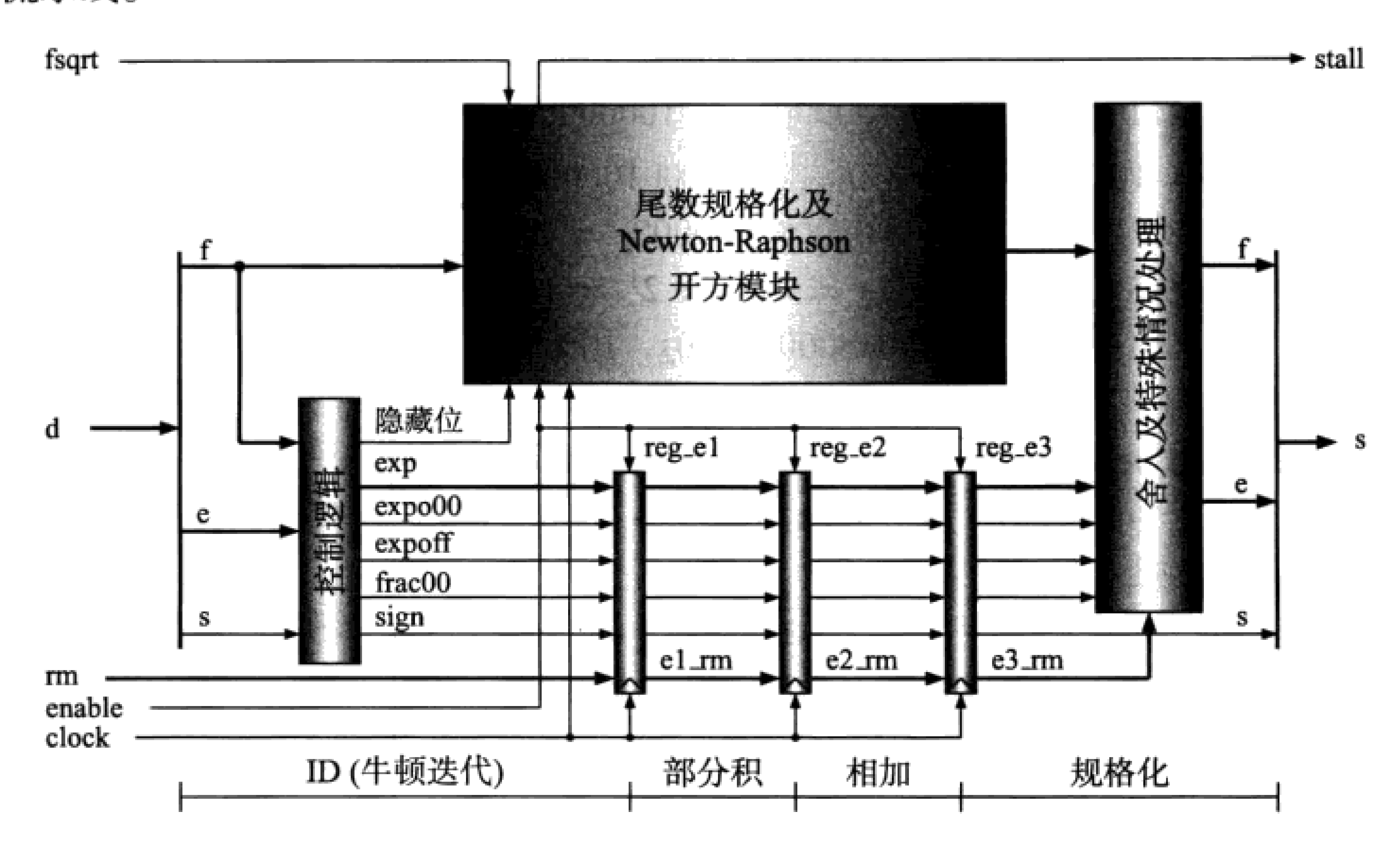


图 9.25 浮点开方电路的总体模块图

图 9.26 是 Newton-Raphson 浮点开方器的流水线示意图。查表得出 x_0 用一个周期，3 次牛顿迭代用 21 个周期。这 22 个周期由 stall 信号暂停流水线。然后是两个周期的乘法和一个周期的规格化。这部分与浮点乘法器类似，用流水线方式实现。将在下面描述如何产生 stall 信号。

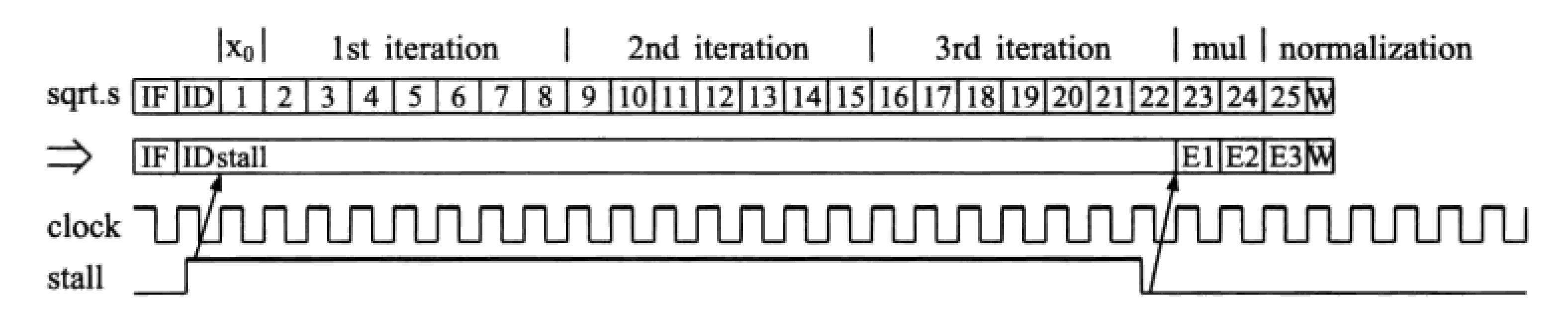


图 9.26 浮点开方指令的流水线

以下是 Newton-Raphson 浮点开方器的 Verilog HDL 代码。

```
module fsqrt_newton (d,rm,fsqrt,enable,clock,resetn,
                     s,busy,stall,count,reg_x);
    input  [31:0] d;        // fp q = root(d)
    input  [1:0]  rm;       // round mode
    input         fsqrt;    // ID stage: fsqrt = i_fsqrt
    input         enable,clock,resetn; // enable
    output [31:0] s;        // fp output
```

```
output        busy;    // for generating stall
output        stall;   // for pipeline stall
output [4:0] count;    // for iteration control
output [25:00] reg_x; // x_i
parameter     ZERO = 31'h00000000;
parameter     INF  = 31'h7f800000;
parameter     NaN  = 31'h7fc00000;
parameter     MAX  = 31'h7f7fffff;
```

以下代码将被用来确定结果是否为特殊浮点数。

```
wire          d_expo_is_00 = ~|d[30:23]; // d_expo = 00
wire          d_expo_is_ff =  &d[30:23]; // d_expo = ff
wire          d_frac_is_00 = ~|d[22:0];  // d_frac = 00
wire          sign = d[31];
```

现在计算临时阶码 exp_8 $= e_d >> 1 + 63 + e_d \% 2$ 及 24 位临时尾数 d_temp24。注意，我们这里假定这全部 24 位已经是小数了，即小数点存在于这 24 位的最左端。我们根据阶码是奇数还是偶数把尾数右移一位或 0 位。如果输入的浮点数是规格化数，则它的格式为 0.1xx···x 或 0.01x···x。

```
wire [7:0]  exp_8 = {1'b0,d[30:24]} + 8'h3f + d[23];
wire [23:0] d_f24 = d_expo_is_00? {d[22:0],1'b0}:{1'b1,d[22:0]};
wire [23:0] d_temp24 = d[23]? {1'b0,d_f24[23:1]}:d_f24; // .01
```

如果输入的浮点数是非规格化数，我们还要把 d_temp24 左移偶数位，使它具有 0.1xx···x 或 0.01x···x 的格式。这项工作由 shift_even_bits 模块完成。我们还要从阶码值中减去移位位数。此时得到的数据有两个：8 位的阶码 exp0 和 24 位的尾数 d_frac24。

```
wire [23:0]  d_frac24;  // to 1xx...x for den
wire [4:0]   shamt_d;   // how many bits shifted
shift_even_bits shift_d (d_temp24,d_frac24,shamt_d);
wire [7:0]   exp0 = exp_8 - {4'h0,shamt_d[4:1]};
```

以下代码实现图 9.25 中的三个流水线寄存器 reg_e1，reg_e2 和 reg_e3。

```
reg e1_sign,e1_e00,e1_eff,e1_f00;
reg e2_sign,e2_e00,e2_eff,e2_f00;
reg e3_sign,e3_e00,e3_eff,e3_f00;
reg [1:0] e1_rm,e2_rm,e3_rm;
reg [7:0] e1_exp,e2_exp,e3_exp;
always @ (negedge resetn or posedge clock)
  if (resetn == 0) begin // pipeline registers
      // reg_e1                // reg_e2              // reg_e3
      e1_sign <= 0;            e2_sign <= 0;          e3_sign <= 0;
      e1_rm   <= 0;            e2_rm   <= 0;          e3_rm   <= 0;
      e1_exp  <= 0;            e2_exp  <= 0;          e3_exp  <= 0;
```

```
        e1_e00  <= 0;             e2_e00  <= 0;         e3_e00  <= 0;
        e1_eff  <= 0;             e2_eff  <= 0;         e3_eff  <= 0;
        e1_f00  <= 0;             e2_f00  <= 0;         e3_f00  <= 0;
    end else if (enable) begin
        e1_sign   <= sign;        e2_sign <= e1_sign;  e3_sign <= e2_sign;
        e1_rm     <= rm;          e2_rm   <= e1_rm;    e3_rm   <= e2_rm;
        e1_exp   <= exp0;         e2_exp  <= e1_exp;   e3_exp  <= e2_exp;
        e1_e00 <= d_expo_is_00;  e2_e00  <= e1_e00;   e3_e00  <= e2_e00;
        e1_eff <= d_expo_is_ff;  e2_eff  <= e1_eff;   e3_eff  <= e2_eff;
        e1_f00 <= d_frac_is_00;  e2_f00  <= e1_f00;   e3_f00  <= e2_f00;
    end
```

以下代码由 root_newton24 模块求尾数的 32 位平方根 frac0，小数点在它的左端。然后我们把它整理成 27 位的 frac = 0.1x· · ·xGRS。注意 frac 中不包括整数部分的 0，小数部分的最左位 1 将成为隐藏位而被扔掉。

```
root_newton24 frac_newton (d_frac24,fsqrt,enable,clock,resetn,
                            frac0,busy,count,reg_x,stall);
wire [31:00] frac0; // root = 1.xxxx...x
wire [26:0]  frac = {frac0[31:6],|frac0[5:0]}; // sticky
```

以下代码对尾数进行舍入，并根据舍入情况调整阶码。得到的结果是 8 位的阶码 e 和 23 位的去掉了隐藏位的尾数 f。阶码不会大于 254，上溢判断似乎没有必要。

```
wire frac_plus_1 = // reg_e3
     ~e3_rm[1] & ~e3_rm[0] & frac[3] & frac[2] & ~frac[1] & ~frac[0] |
     ~e3_rm[1] & ~e3_rm[0] & frac[2] & (frac[1] | frac[0]) |
     ~e3_rm[1] &  e3_rm[0] & (frac[2] | frac[1] | frac[0]) & e3_sign |
      e3_rm[1] & ~e3_rm[0] & (frac[2] | frac[1] | frac[0]) & ~e3_sign;
wire [24:0]  frac_round = {1'b0,frac[26:3]} + frac_plus_1;
wire [7:0]   exp1 = frac_round[24]? e3_exp + 8'h1 : e3_exp;
wire         overflow = (exp1 >= 10'h0ff); // overflow
```

我们还要对输入的特殊浮点数进行处理，这部分工作由函数 final_result 完成。由此我们得到最后结果 s。

```
wire [7:0]   exponent;
wire [22:0]  fraction;
assign {exponent,fraction} = final_result(overflow,e3_rm,e3_sign,
       e3_sign,e3_e00,e3_eff,e3_f00,{exp1[7:0],frac_round[22:0]});
assign s = {e3_sign,exponent,fraction}; // reg_e3
```

函数 final_result 对特殊浮点数进行处理。处理结果有 4 种可能：无穷大，0，NaN 或者计算出的结果。上溢不会出现，相关代码可以删除。

```
function [30:0] final_result;
    input overflow;
    input [1:0] e3_rm;
    input e3_sign;
```

```
        input d_sign,d_e00,d_eff,d_f00;
        input [30:0] calc;
        casex ({overflow,e3_rm,e3_sign,d_sign,d_e00,d_eff,d_f00})
            8'b100x_xxxx : final_result = INF;  // overflow
            8'b1010_xxxx : final_result = MAX;  // overflow
            8'b1011_xxxx : final_result = INF;  // overflow
            8'b1100_xxxx : final_result = INF;  // overflow
            8'b1101_xxxx : final_result = MAX;  // overflow
            8'b111x_xxxx : final_result = MAX;  // overflow
            8'b0xxx_1xxx : final_result = NaN;  // negative
            8'b0xxx_0010 : final_result = NaN;  // nan
            8'b0xxx_0011 : final_result = INF;  // inf
            8'b0xxx_0101 : final_result = ZERO; // 0
            8'b0xxx_000x : final_result = calc; // normal
            8'b0xxx_0100 : final_result = calc; // den
            default      : final_result = NaN;
        endcase
    endfunction
endmodule
```

以下代码就是前面提到的 shift_even_bits 模块，专门应付非规格化浮点数。

```
module shift_even_bits (a,b,shamt);
    input  [23:00] a; // shift a=xxx...x to b=1xx...x or 01x...x
    output [23:00] b; // 1x...x
    output [4:0] shamt; // how many bits shifted
    wire [23:0]  a5,a4,a3,a2,a1;
    assign a5 = a;
    assign shamt[4] = ~|a5[23:08];                    // 16-bit 0
    assign a4 = shamt[4]? {a5[07:00],16'b0} : a5;
    assign shamt[3] = ~|a4[23:16];                    //  8-bit 0
    assign a3 = shamt[3]? {a4[15:00], 8'b0} : a4;
    assign shamt[2] = ~|a3[23:20];                    //  4-bit 0
    assign a2 = shamt[2]? {a3[19:00], 4'b0} : a3;
    assign shamt[1] = ~|a2[23:22];                    //  2-bit 0
    assign a1 = shamt[1]? {a2[21:00], 2'b0} : a2;
    assign shamt[0] = 0;
    assign b = a1;
endmodule
```

对 24 位尾数求平方根与我们在第 3 章描述的对 32 位小数求平方根的 Newton-Raphson 算法类似，不同之处是最后的乘法用流水线方式实现 (见图 9.27) 以及使用了实际的时钟周期来计算何时出来最后结果。图 9.27 中的 reg_m 和 reg_q 是两级流水线寄存器。在迭代完成后，立即进行部分积的运算并把结果打入 reg_m 寄存器。

```
module root_newton24 (d,fsqrt,enable,clock,resetn,q,busy,count,
                      reg_x,stall);
```

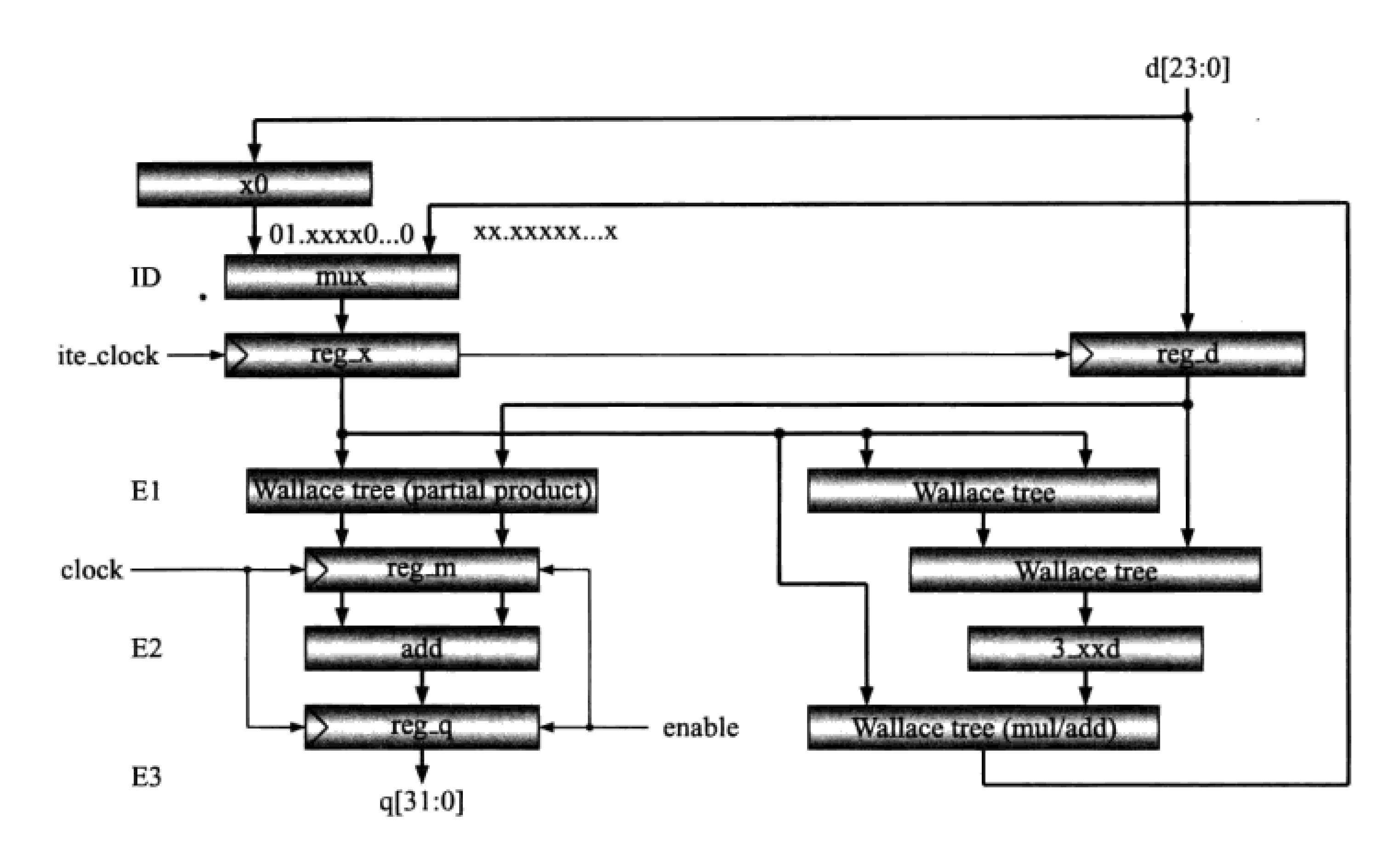


图 9.27 Newton-Raphson 开方电路

```
input [23:00] d;  // radicand: fraction: .1xxx...x or .01xx...x
input         fsqrt;  // ID stage: fsqrt = i_fsqrt
input         enable,clock,resetn;
output [31:00] q;     // root:     .1xxx...x
output         busy;  // cannot receive new div
output         stall; // stall to save result
output [4:0]   count; // for sim test only
output [25:00] reg_x; // for sim test only 01.xx...x
reg [31:00]    q;     // root:     .1xxx...x
reg [23:00]    reg_d; // 24-bit:   .xxxx...xx
reg [25:00]    reg_x; // 26-bit: xx.1xxx...xx
reg [4:0]      count; // 3 iterations
reg            busy;
```

初始值 x_0 通过查 ROM 表得到。当 start 信号有效时，我们对寄存器赋初值。与我们在第 3 章介绍的 32 位版本不同，我们这里使用了实际的时钟周期进行计数，而不是对迭代次数进行计数。Newton-Raphson 的迭代公式为 $x_{i+1} = x_i(3 - x_i^2 d)/2$。当 x_3 计算出之后，下一个周期使用 Wallace 树型乘法算法计算出部分积的和 m_s 与进位 m_c，并把它们保存在 reg_m 寄存器中。一次迭代用 7 个时钟周期：三次乘法用 6 个周期，一次减法用一个周期。在计算过程中，当计数值为 8，15 和 22 时，我们修改寄存器 x_i 的内容 (3 次迭代)。迭代完成后还要两个周期实现 $d \times x_3$。最后的规格化还要一个周期，总共需要 25 个周期。

```
wire [7:0]     x0 = rom(d[23:19]);
always @ (posedge clock or negedge resetn) begin
    if (resetn == 0) begin
```

```
            count <= 5'b0; // reset count
            busy  <= 0;    // reset to not busy
        end else begin     // not reset
            if (fsqrt & (count == 0)) begin // do once only
                count <= 5'b1;              // set count
                busy  <= 1'b1;              // set to busy
            end else begin  // execution: 3 iterations
                if  (count == 5'h01) begin
                     reg_x <= {2'b1,x0,16'b0}; // 01.xxxx0...0
                     reg_d <= d;               //   .1xxxx...x
                end
                if  (count != 0) count <= count + 5'b1; // count++
                if  (count == 5'h15) busy <= 0;  // ready for next
                if  (count == 5'h16) count <= 5'b0; // reset count
                if ((count == 5'h08) ||
                    (count == 5'h0f) ||
                    (count == 5'h16))
                     reg_x <= x52[50:25];     // /2 = xx.xxxxx...x
            end
        end
    end
```

我们使用简单的暂停流水线的方法等待浮点开方指令的执行。下面的 stall 信号为 1 时，流水线暂停。我们希望它的波形如图 9.28 所示，因此我们产生暂停流水线的信号 stall 如下：

```
assign stall = fsqrt & (count == 0) | busy;
```

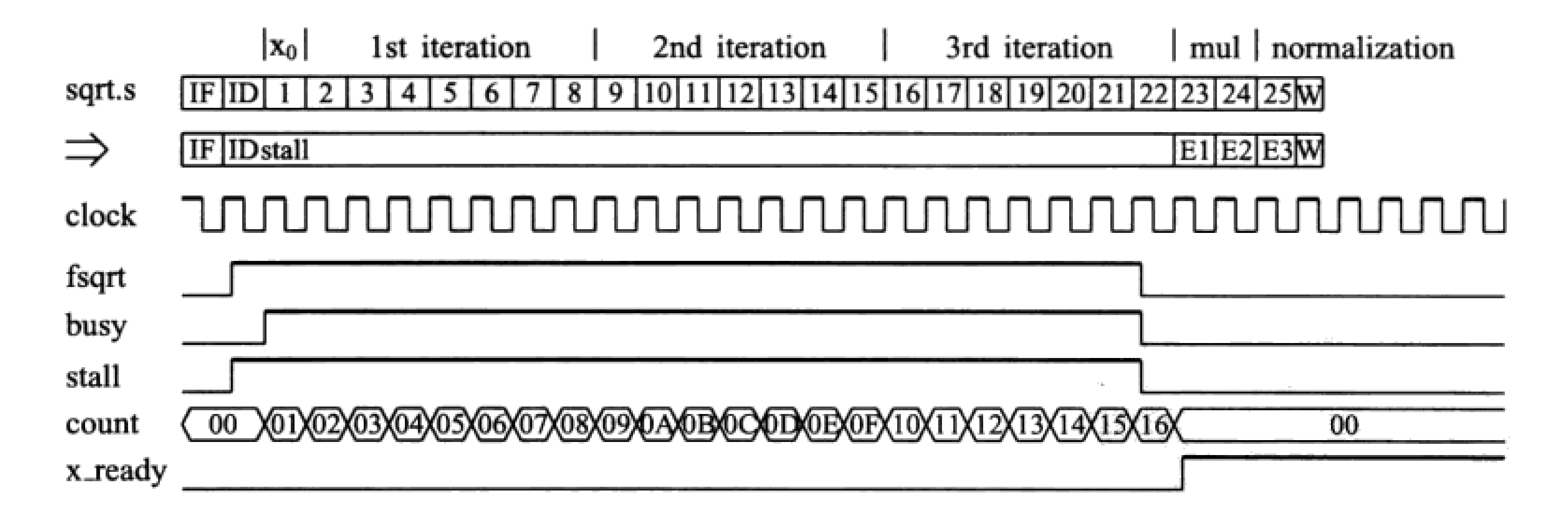


图 9.28 浮点开方指令暂停流水线的信号 stall 的产生

以下代码实现 $x_{i+1} = x_i(3 - x_i^2 d)/2$。乘法用 Wallace 树型算法实现。

```
wire [51:00] x_2;                          // xxxx.xxxxx...x
wire [51:00] x2d;                          // xxxx.xxxxx...x
wire [51:00] x52;                          // xxxx.xxxxx...x
```

```
    wallace_tree26x26 x2 (reg_x,reg_x,x_2);
    wallace_tree24x28 xd (reg_d,x_2[51:24],x2d);
    wire [25:00] b26 = 26'h3000000 - x2d[49:24]; //   xx.xxxxx...x
    wallace_tree26x26 xip1 (reg_x,b26,x52);
```

如果 3 次迭代完成，我们用一次乘法 ($d \times x_3$) 计算出 32 位的平方根。乘法用 Wallace 树型算法及流水线方式实现，即在部分积和加法器之间加了一个流水线寄存器。两级流水线寄存器 reg_m 和 reg_q 分别存放部分积 (m_s 和 m_c) 和相加的结果 (e2p)。我们这里没有对商进行舍入，因为舍入在规格化阶段进行。

```
    wire [48:0]  m_s; // 41 + 8 = 49-bit
    wire [41:0]  m_c; // 42-bit
    wallace24x26_mul wt (reg_d,reg_x,m_s[48:8],m_c,m_s[7:0]);
    reg [48:0]   a_s; // 41-bit
    reg [41:0]   a_c; // 42-bit
    always @ (negedge resetn or posedge clock)
      if (resetn == 0) begin // registers: reg_m and reg_q
          a_s <= 0; // reg_m
          a_c <= 0; // reg_m
          q   <= 0; // reg_q
      end else if (enable) begin
          a_s <= m_s; // save partial product sum
          a_c <= m_c; // save partial product carry
          q   <= e2p;
      end
    wire [49:00] d_x = {1'b0,a_s} + {a_c,8'b0}; // 0x.xxxxx...x
    wire [31:00] e2p = {d_x[47:17],|d_x[16:0]}; // sticky
    function [7:0] rom; // 1/d^{1/2}
        input [4:0] d;
        case (d)
            5'h08: rom = 8'hf0;              5'h09: rom = 8'hd5;
            5'h0a: rom = 8'hbe;              5'h0b: rom = 8'hab;
            5'h0c: rom = 8'h99;              5'h0d: rom = 8'h8a;
            5'h0e: rom = 8'h7c;              5'h0f: rom = 8'h6f;
            5'h10: rom = 8'h64;              5'h11: rom = 8'h5a;
            5'h12: rom = 8'h50;              5'h13: rom = 8'h47;
            5'h14: rom = 8'h3f;              5'h15: rom = 8'h38;
            5'h16: rom = 8'h31;              5'h17: rom = 8'h2a;
            5'h18: rom = 8'h24;              5'h19: rom = 8'h1e;
            5'h1a: rom = 8'h19;              5'h1b: rom = 8'h14;
            5'h1c: rom = 8'h0f;              5'h1d: rom = 8'h0a;
            5'h1e: rom = 8'h06;              5'h1f: rom = 8'h02;
            default: rom = 8'hff;
        endcase
    endfunction
endmodule
```

浮点开方总共需要 25 个周期：查表用一个周期，3 次迭代 21 个，最后的乘法两个，规格化一个。但浮点开方指令的发出只需隔 22 个周期，因为最后的三个周期使用流水线方式。开方运算比除法运算多用了 6 个周期：3 次迭代每次多用两个 (多一次乘法)。图 9.29 给出了我们在本节开始处举的两个例子的仿真结果，即 $d = 41100000_{16}$，$s = 40400000_{16}$ 和 $d = 00003200_{16}$，$s = 1DA00000_{16}$。

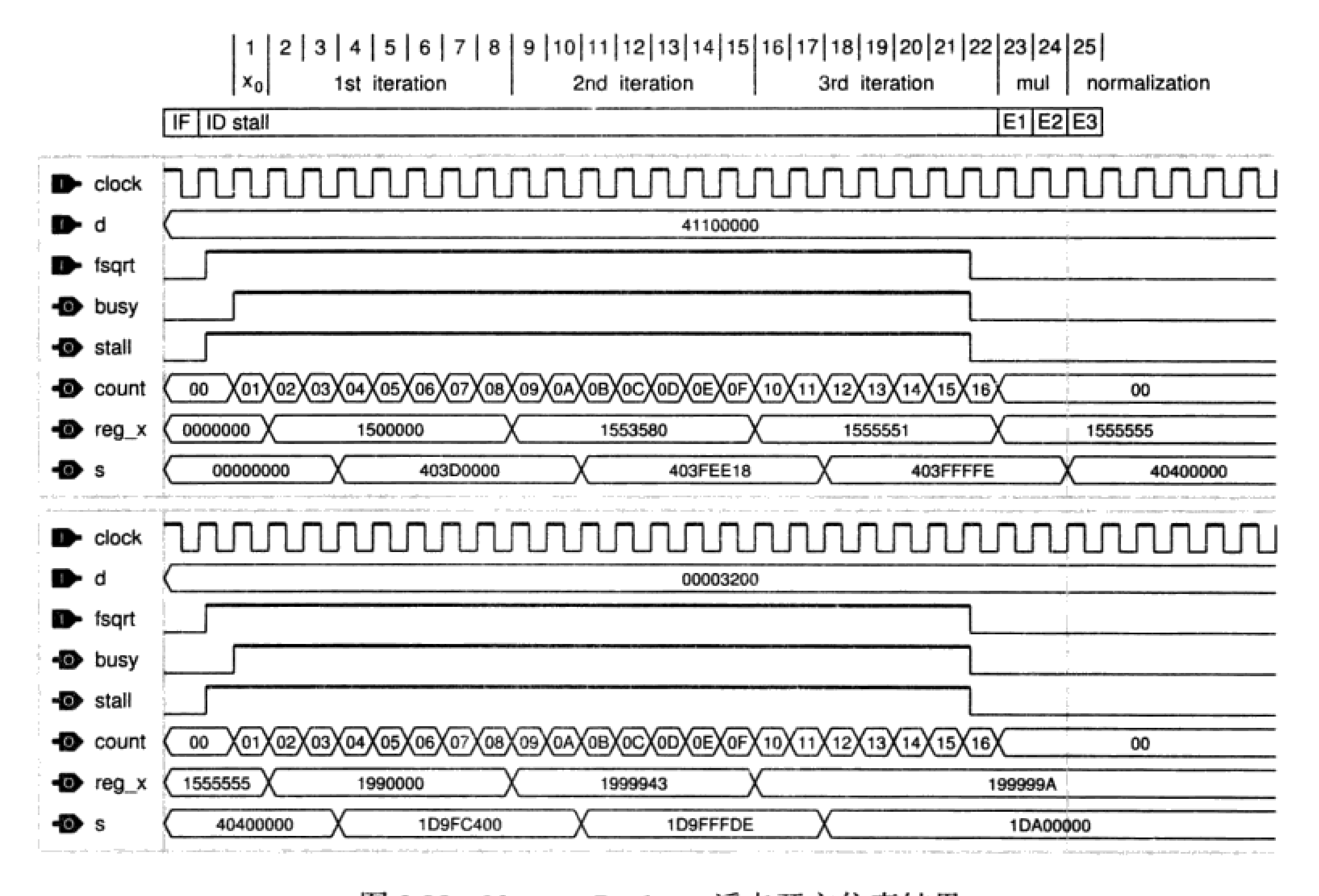


图 9.29 Newton-Raphson 浮点开方仿真结果

9.7 习题

1. 本章给出的浮点数与整数之间的转换代码没有考虑舍入方式。试加入舍入方式信号 rm[1:0]，重新编写 Verilog HDL 代码。
2. 试设计双精度浮点数与整数及单精度浮点数之间的转换电路。
3. 试设计一个 5 级流水线的浮点加法器：阶码对齐用两级，计算用一级，规格化用两级。
4. 试编写 24 位乘 26 位的 Wallace 树型乘法器的 Verilog HDL 代码。
5. 手工编写 Wallace 树型乘法器的 Verilog HDL 代码似乎太头疼了。试用高级语言编写一个程序，自动生成 Verilog HDL 代码。被乘数和乘数的位数应该是变量 (这个程序应该是一个很有用工具，最好用 Script 语言编写，不用编译，比如 Perl 或 Python)。

6. 试使用 Goldschmidt 算法设计浮点除法电路。
7. 试使用 Goldschmidt 算法设计浮点开方电路。
8. 试探讨实现流水线浮点除法及开方电路的可能性，即每个时钟周期能接收一条除法或开方指令。

第 10 章　带有 FPU 的流水线 CPU 及其 Verilog HDL 设计

本章描述带有 FPU 的流水线 CPU 的设计方法。浮点部件可执行的指令有加减乘除及开方。浮点乘法使用 Wallace 树型算法。浮点除法和开方使用 Newton-Raphson 算法。由于流水线操作有数据相关的问题并且浮点运算指令所需的流水线级数不同，该 CPU 设计起来要比单纯的只有整数部件的流水线 CPU 复杂一些。

我们在第 8 章的整数流水线 CPU 的基础上，增加浮点运算指令和浮点寄存器堆与存储器之间的数据传送指令。本章的重点是如何处理多个执行周期的浮点指令。我们给出 CPU 的总体结构，Verilog HDL 代码以及 CPU 的仿真波形图。

10.1　CPU / FPU 流水线模型

本节我们首先简要介绍 CPU 可执行的与浮点操作有关的指令，然后介绍实现这些指令时所采用的流水线模型。

10.1.1　CPU / FPU 可执行的指令

CPU 可执行的指令共有 27 条，其中 20 条是整数指令，与第 8 章流水线 CPU 实现的指令相同。新加的指令有 7 条：浮点运算指令共有 5 条，它们是单精度的浮点加 add.s、浮点减 sub.s、浮点乘 mul.s、浮点除 div.s 和浮点开方 sqrt.s。另外的两条是从存储器取数据写入浮点寄存器堆的指令 lwc1 和把浮点寄存器堆的数据写入存储器的指令 swc1。MIPS 把 FPU 称做协处理机 1 (Coprocessor 1)，因此在指令助记符中加了 c1。以下我们简要介绍新加的 7 条指令。浮点加减乘除指令具有相同的格式：

```
add.s/sub.s/mul.s/div.s fd, fs, ft  # fd <-- fs op ft;
```

其中，fs 和 ft 是两个源操作数的浮点寄存器号，fd 是浮点目的寄存器号。三个寄存器号都用 5 位二进制数表示，这意味着浮点寄存器堆有 $2^5 = 32$ 个浮点寄存器。我们用 f0, f1, $\cdots$, f31 表示这 32 个寄存器。注意，整数寄存器堆中的 r0 的内容永远是 0，而浮点寄存器堆中的 f0 是一个普通的寄存器。浮点开方的指令格式如下：

```
sqrt.s fd, fs  # fd <-- root (fs);
```

其中的 fs 是源操作数的寄存器号，fd 是目的寄存器号。存储器访问指令 lwc1 和 swc1 分别与 lw 和 sw 类似，它们的格式如下：

```
lwc1 ft, offset(rs)  # ft <-- memory[rs + offset];
swc1 ft, offset(rs)  # memory[rs + offset] <-- ft;
```

指令 lwc1 从整数寄存器 rs 中读出 32 位数据，与符号扩展的 offset 相加，得到的结果作为存储器地址，从存储器中读出 32 位数据，写入浮点寄存器 ft 中。swc1 与 lwc1 相反，它把浮点寄存器 ft 的内容写入存储器，存储器地址的计算方法与 lwc1 相同。表 10.1 列出了这 7 条指令的格式。

表 10.1 与 FPU 有关的 7 条指令的格式

指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	意 义
lwc1	110001	rs	ft	offset			取浮点存储器字
swc1	111001	rs	ft	offset			存浮点存储器字
add.s	010001	10000	ft	fs	fd	000000	浮点加
sub.s	010001	10000	ft	fs	fd	000001	浮点减
mul.s	010001	10000	ft	fs	fd	000010	浮点乘
div.s	010001	10000	ft	fs	fd	000011	浮点除
sqrt.s	010001	10000	00000	fs	fd	000100	浮点开方

10.1.2 CPU/FPU 基本的流水线模型

指令 lwc1 和 swc1 的流水线与 lw 和 sw 类似，但浮点运算指令的流水线要复杂一些。基本的流水线模型如图 10.1 所示。

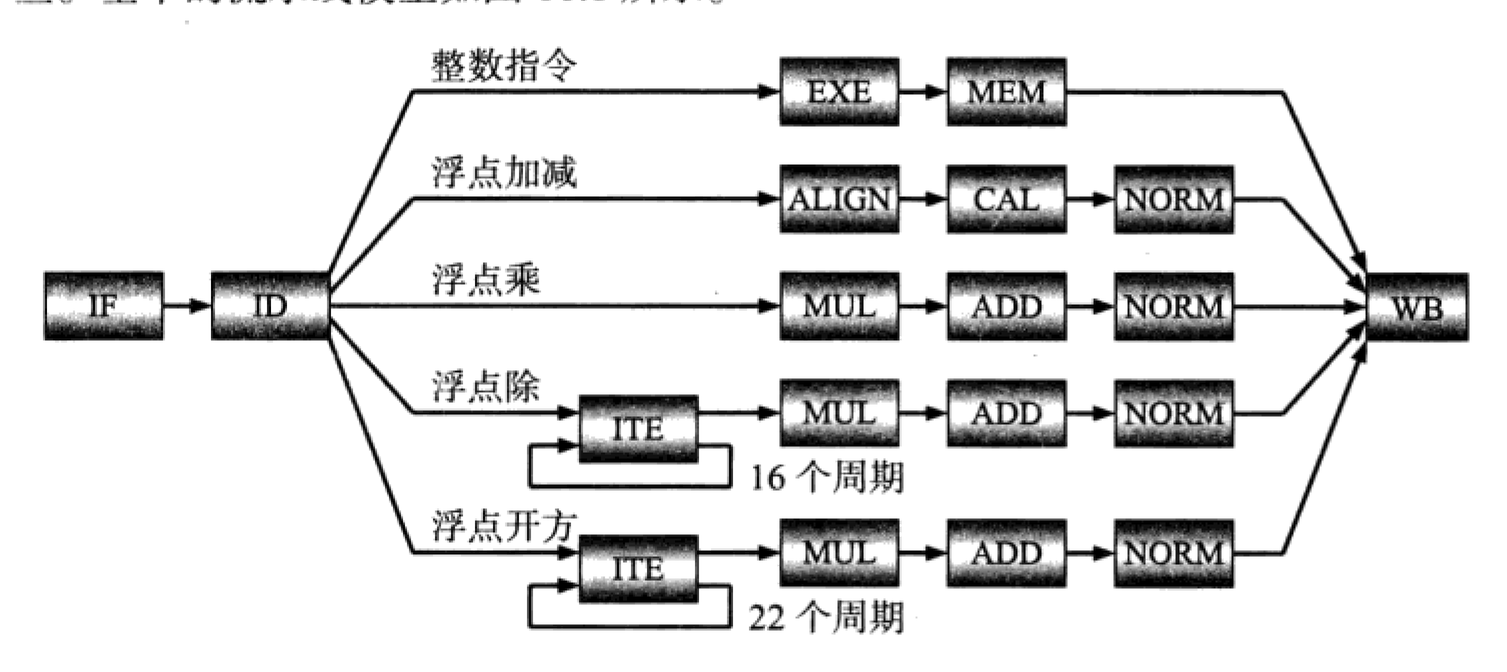


图 10.1 CPU/FPU 流水线模型

整数指令的流水线与第 8 章描述的流水线 CPU 相同，有 5 级。浮点加、减指令有 6 级，其中 ALIGN 完成阶码对齐，CAL 对浮点尾数进行计算，NORN 完成浮点数的规格化。浮点乘法指令的流水线也有 6 级，其中 MUL 是用 Wallace 树型算法计算部分积，ADD 级把部分积相加，NORN 完成浮点数的规格化。浮点加、减、乘指令可以用流水线的方式执行，即每个周期可以接收一条指令。浮点除法和浮点开方指令的执行有些复杂。在 ITE (迭代) 级，我们用 Newton-Raphson 的迭代算法

计算出 1/b (除法指令) 和 $1/\sqrt{d}$ (开方指令)。然后再用一次乘法，计算出商和平方根：$q = a \times (1/b) = a/b$、$q = d \times (1/\sqrt{d}) = \sqrt{d}$。这部分与浮点乘法的 MUL 及 ADD 相同，使用 Wallace 树型算法。然后是规格化级。浮点除法指令的 ITE 级需要 16 个周期：查 ROM 表需要一个周期，3 次迭代需要 15 个周期。浮点开方指令的 ITE 级需要 22 个周期：查 ROM 表需要一个周期，3 次迭代需要 21个周期，见图 10.2。

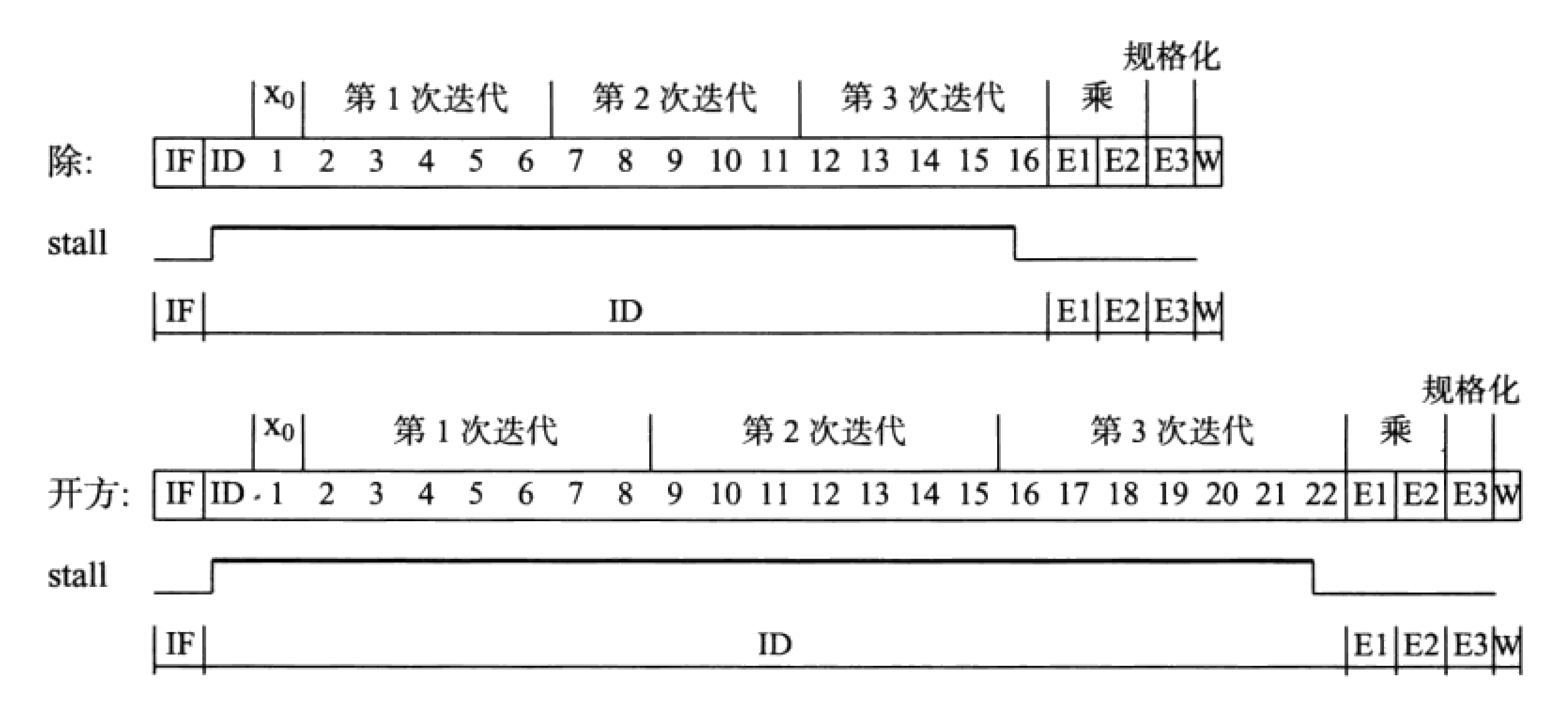


图 10.2 浮点除法和开方指令的流水线

为了使电路的实现变得简单，我们统一了浮点加减乘除和开方的流水线模型，见图 10.3。对浮点除法和开方指令，我们把 ITE 与 ID 级合并为一级，在此期间暂停流水线的取指令操作。这样，所有的浮点指令就有了统一的流水线模型：IF、ID、E1、E2、E3、WB。在以后的流水线数据相关问题的讨论中，我们就使用这个流水线模型，不再区分是否有迭代。

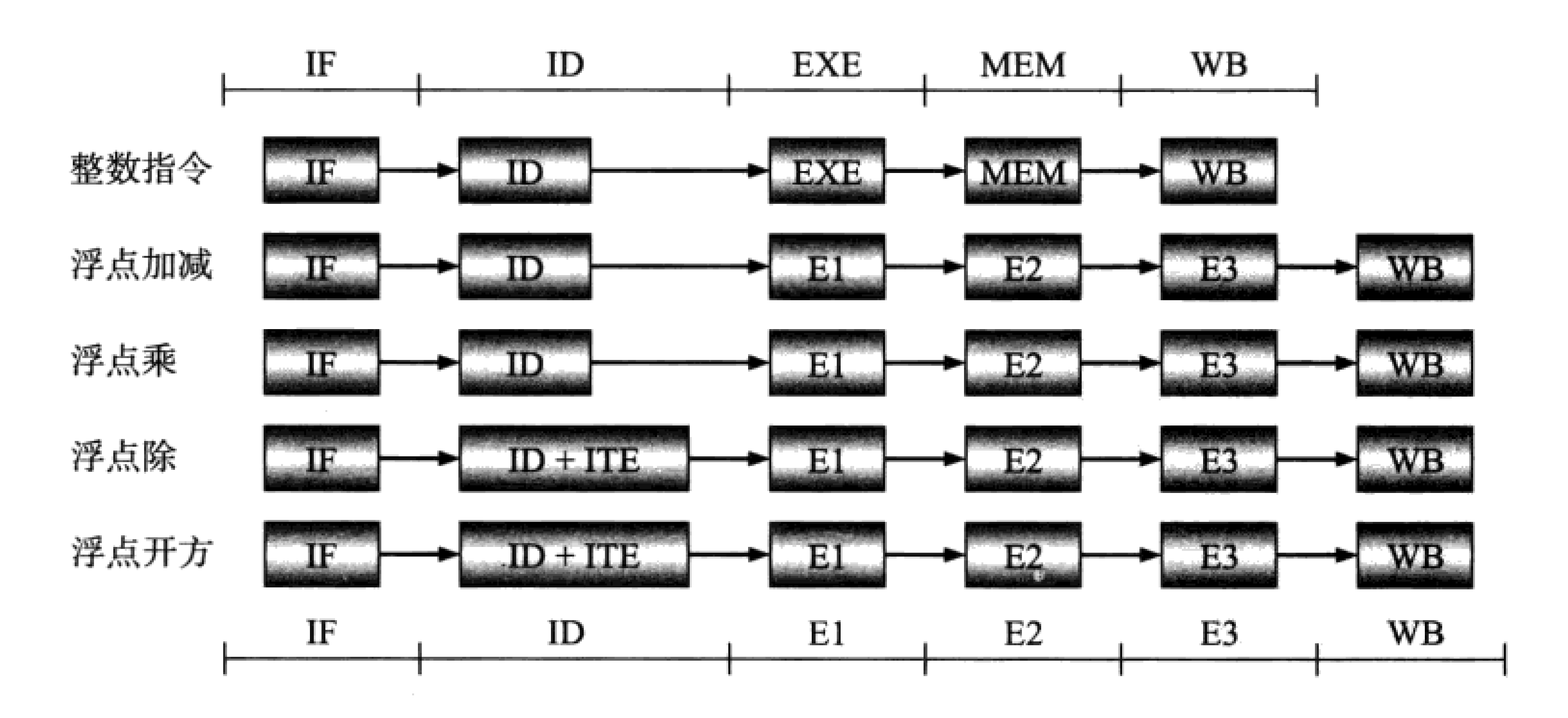


图 10.3 统一的流水线模型

10.2 带有两个写端口的寄存器堆设计

浮点寄存器堆有 32 个浮点寄存器。与整数部件的寄存器堆不同，浮点寄存器堆需要有两个写端口。为什么？看看图 10.4 就知道了。由于浮点指令的流水线有 6 级，而 lwc1 指令的流水线只有 5 级，出现了在同一个周期有两个向浮点寄存器堆的写入操作：一个是浮点运算指令的结果写回，另一个是存储器数据向浮点寄存器堆的写入。因此我们需要设计一个带有两个写端口的寄存器堆。

指令									
lwc1 f0, 100(r1)	IF	ID	EXE	MEM	WB				
lwc1 f1, 100(r2)		IF	ID	EXE	MEM	WB			
add.s f2, f2, f0			IF	ID	E1	E2	E3	WB	
sub.s f3, f3, f1				IF	ID	E1	E2	E3	WB
lwc1 f4, 100(r3)					IF	ID	EXE	MEM	WB

两个写操作

图 10.4 需要两个写端口的寄存器堆

图 10.5 所示的是两个写端口的寄存器堆的逻辑电路图，其中左上角部分是寄存器堆的电路符号，其余部分是寄存器堆的详细内部电路。

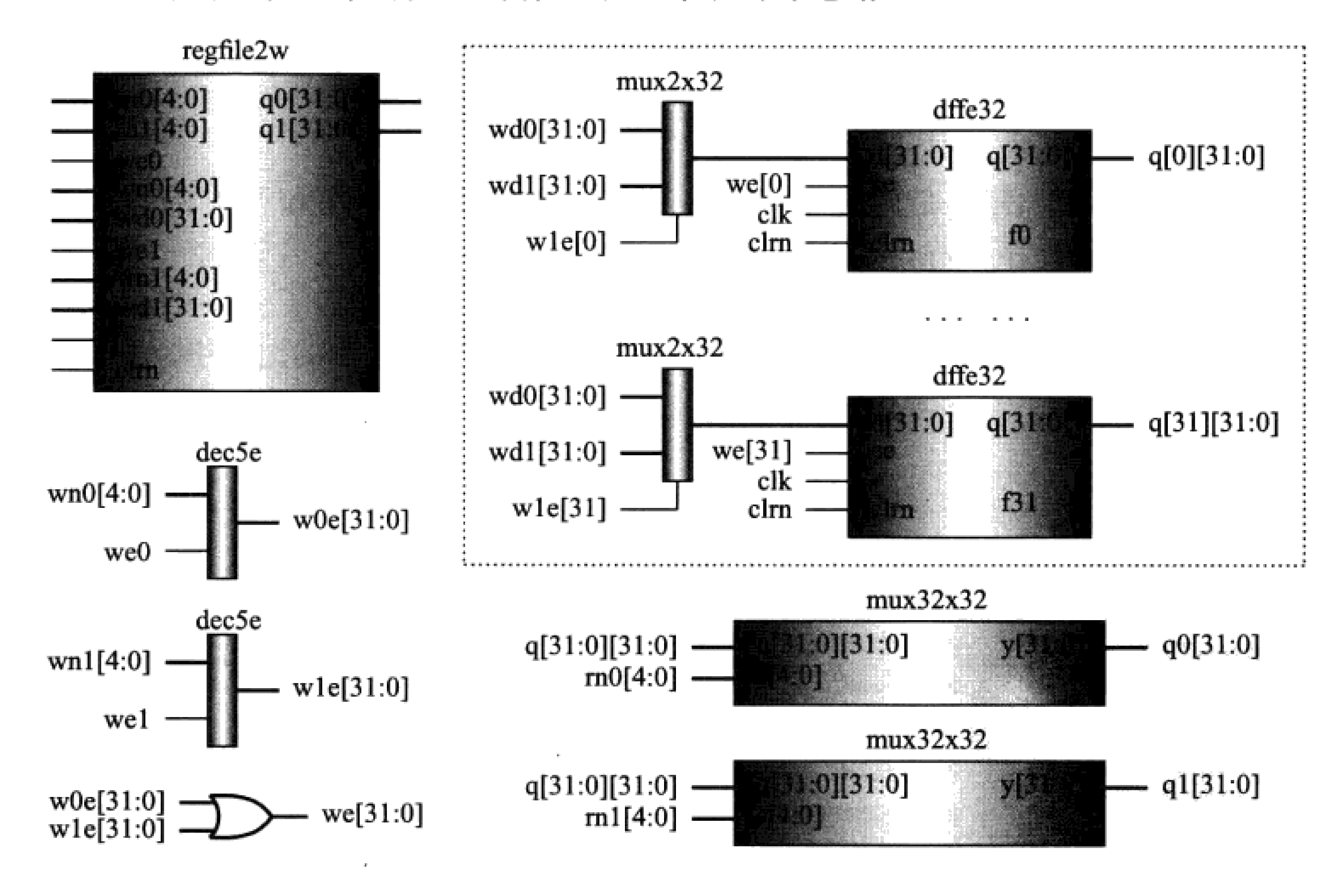


图 10.5 两个写端口的寄存器堆的详细电路

图中右上角较大的虚线框中是 32 个 32 位的寄存器 (32 个 dffe32) 和 32 个二选一多路器 (每个寄存器一个，用于选择向寄存器写入的数据)。寄存器堆的读端口与整数部件的寄存器堆相同，根据两个 5 位的寄存器号 rn0 和 rn1，从寄存器堆中选择两个

32 位的寄存器，把它们所保存的内容由 q0 和 q1 端口分别送出 (图中的右下部分)。这部分电路由两个简单的多路选择器 (mux32x32) 实现。

两个写端口的编号分别为 0 和 1。当写使能信号 we0 (we1) 为 1 时，32 位的数据 wd0 (wd1) 在时钟的上升沿处写入 wn0 (wn1) 号寄存器。对两个寄存器号 wn0 和 wn1 的译码分别由两个 5-32 译码器 (dec5e) 实现。译码器带有使能端，为 0 时，译码器的所有 32 位输出均为 0。只有当使能端为 1 时，译码器的 32 位输出中才有一位为 1，其余 31 位为 0。两个译码器相应的输出位相“或”后，分别接到各自的寄存器 dffe32 的使能端。即只要有一个写端口选中一个寄存器，数据就要被写入其中。

如果两个写端口同时要向同一个寄存器写入数据，问题就来了。又不能两个都写，总要选一个。我们这里用了最简单的方法：1 号写端口的优先级比 0 号高，选 1 号，丢掉 0 号。寄存器 dffe32 的 d 输入端前面的二选一多路器 mux2x32 就是干这个的，选择端由 1 号控制。如果两个写端口分别向不同的寄存器写入数据，大家相安无事。在编写汇编程序时，应能避免向同一寄存器的同时写入。

以下就是它的 Verilog HDL 代码，比电路图简单多了。不难读懂，只是要注意代码中把写端口的 0 号和 1 号分别用 x 和 y 来代替，而把读端口的 0 号和 1 号分别又用 a 和 b 来代替了。读懂后你会不会感觉 Verilog HDL 还不错?

```
module regfile2w (rna,rnb,dx,wnx,wex,dy,wny,wey,clk,clrn,qa,qb);
    input   [4:0] rna,rnb,wnx,wny;
    input  [31:0] dx,dy;
    input         wex,wey,clk,clrn;
    output [31:0] qa,qb;
    reg [31:0]    register [0:31];
    assign        qa = register[rna]; // read port a
    assign        qb = register[rnb]; // read port b
    always @(posedge clk or negedge clrn)
        if (clrn == 0) begin
            integer i;
            for (i = 0; i < 32; i = i + 1) register[i] <= 0;
        end else begin
            if (wey) // write port y has a higher priority than x
                register[wny] <= dy; // write port y
            if (wex && (!wey || (wnx != wny)))
                register[wnx] <= dx; // write port x
        end
endmodule
```

10.3 浮点数据相关以及流水线暂停

本节讨论浮点数据相关及流水线暂停的问题，包括三方面内容：(1) 浮点运算指令之间的数据相关；(2) 存储器访问指令与浮点运算指令之间的数据相关；(3) 浮点除法和开方指令如何暂停流水线。图 10.6 是一个简化的 CPU 内部结构图，上半部分是浮点部件 FPU，下半部分是整数部件 IU。各流水线级之间使用了流水线寄存器。

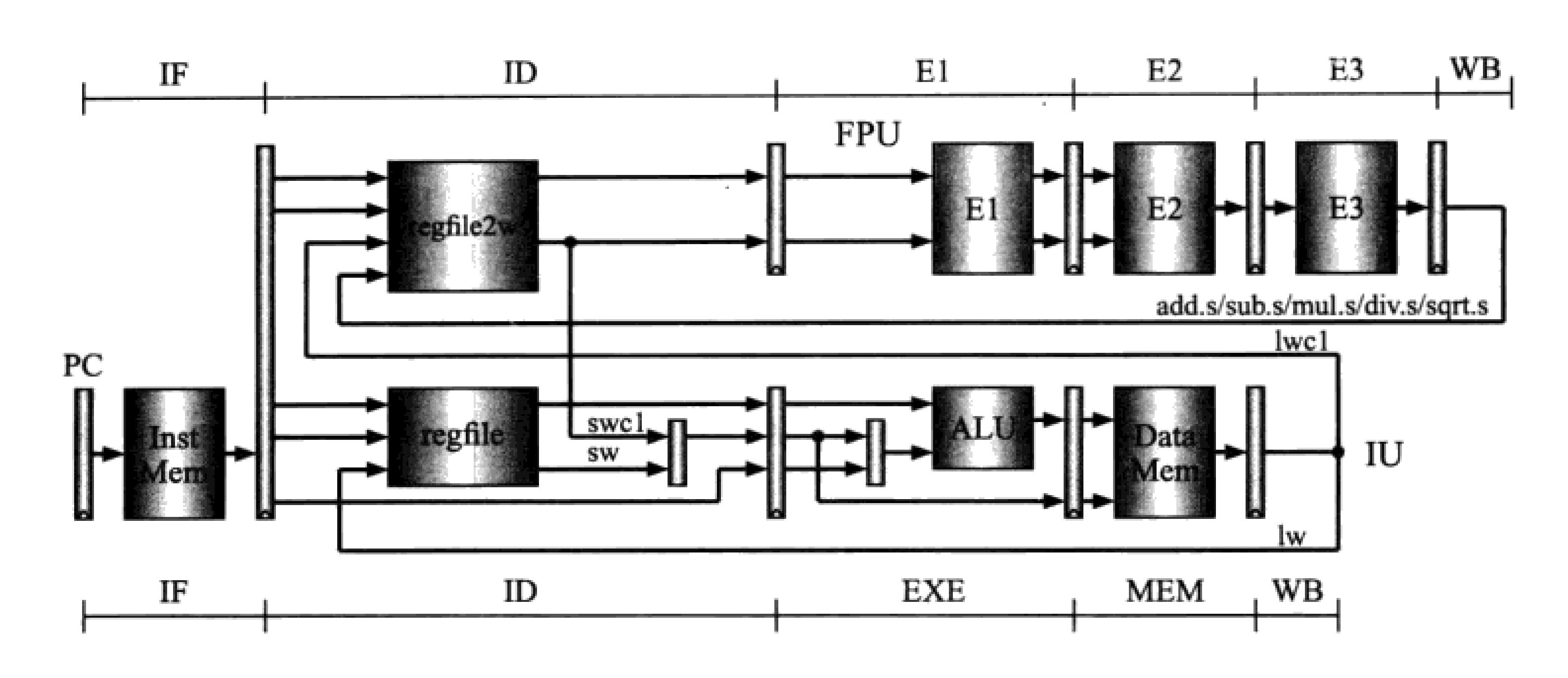


图 10.6　简化的带有 FPU 的 CPU 结构

注意图中没有给出任何内部前推的电路，浮点除法和开方指令的流水线模型也被简化成与浮点加减乘指令相同。该图是基础，通过本节的讨论，我们慢慢把所需的电路加到它上面，从而得到最终的电路。

10.3.1　浮点运算结果的内部前推和流水线暂停

浮点运算指令 (add.s, sub.s, mul.s, div.s 和 sqrt.s) 之间的数据相关要比整数运算指令之间的数据相关复杂得多。整数运算指令在 EXE 级结束时就能得到结果 (不包括 lw 指令)，只要使用数据内部前推就能解决数据相关问题，不用停流水线。浮点指令要在 E3 级结束时才能得到结果，因此可能要停流水线。

首先从最简单的情况开始讨论数据相关问题。见图 10.7，第 1 条指令 add.s f2, f1, f0 把寄存器 f1 和 f0 的内容相加，结果送入 f2。流水线级 IF、ID、E1、E2、E3、WB 分别用 add.s、f1,f0、e1、e2、e3、f2 来表示：ID 级 (第 2 个周期) 从寄存器堆读出寄存器 f1 和 f0 的内容，WB 级 (第 6 个周期的前半部分) 把加法结果写入寄存器 f2。

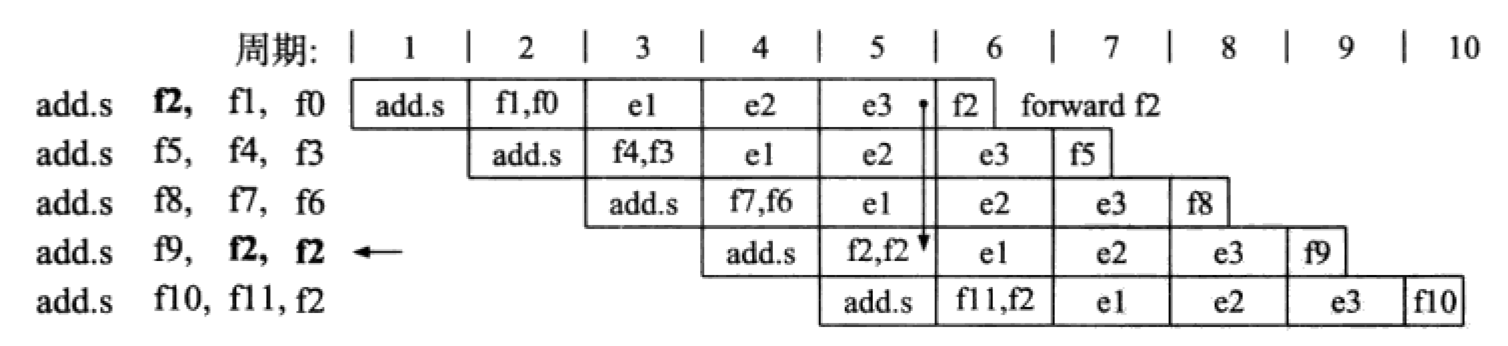


图 10.7　浮点数据的内部前推、不需停流水线

图中最后一条指令没有问题，它可以在 ID 级的后半部分读出 f2 的内容。第 4 条指令 add.s f9, f2, f2 使用第一条指令的结果。注意第 4 条指令在第 5 个周期就要读寄存器 f2 的内容，这时第 1 条指令还没有把数据写入 f2。这个数据相关问题可以使用内部前推技术来解决，如图 10.7 中的箭头线所示。

为此，我们在 ID 级配备两个二选一多路器，选择信号分别为 fwdfa 和 fwdfb。当图 10.7 的情况出现时，选择 E3 级的输出，而不是寄存器堆的输出。在 ID 级结束时的上升沿处，把选中的数据打入流水线寄存器，如图 10.8 所示。图中的 fop 是浮点操作控制码，fwfpr 是浮点运算指令的结果写回信号 (浮点目的寄存器的写使能)，fd 是浮点指令中的目的寄存器号。

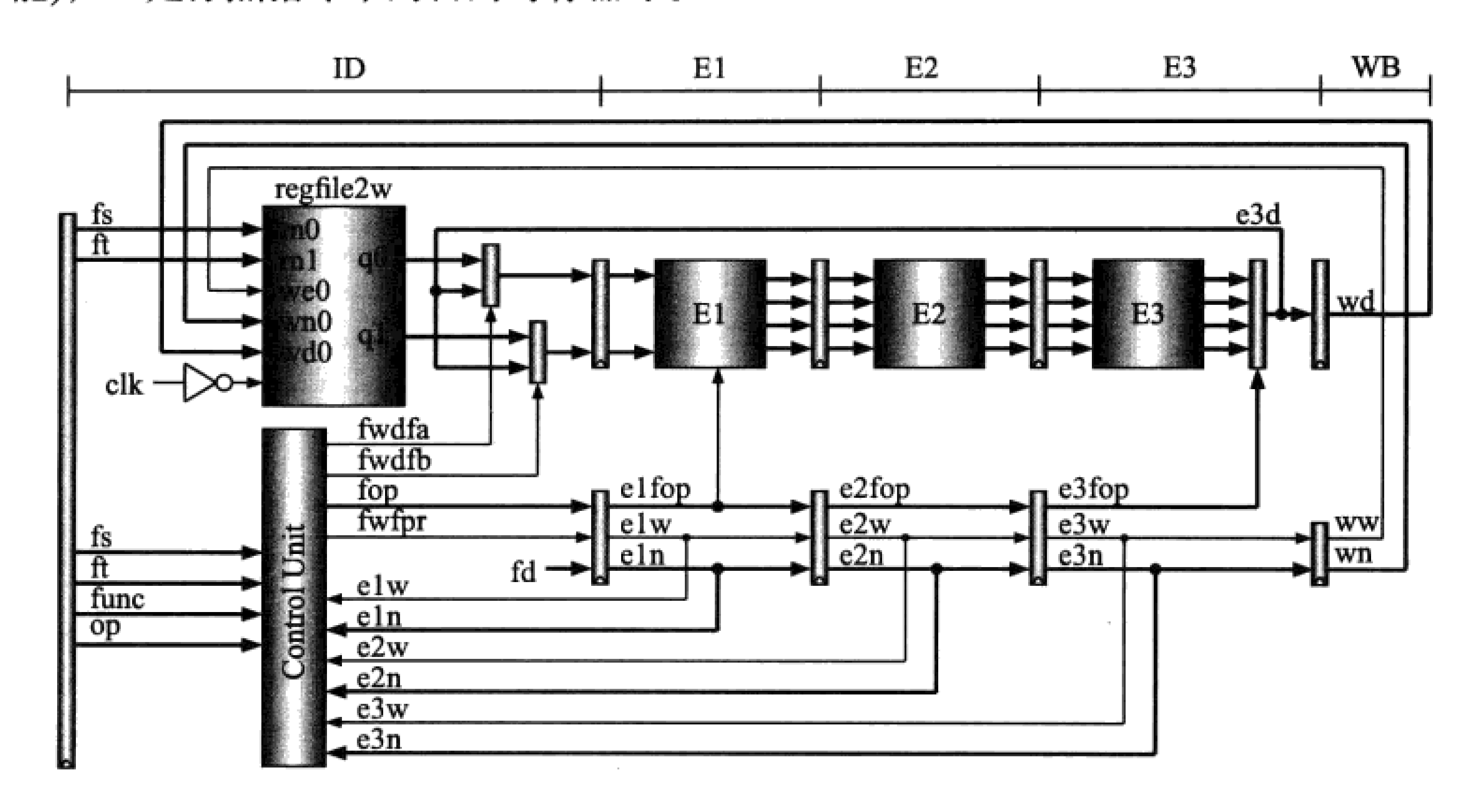


图 10.8 浮点部件的内部前推电路

两个二选一多路器的选择信号如何产生是问题的关键。从前面的讨论可知，如果处在 E3 级的指令是一条浮点运算指令，并且它的目的寄存器号与 ID 级指令的一个源寄存器号相同，则要选择 E3 级的数据。因此，两个多路器的选择信号分别为：

```
fwdfa = e3w & (e3n == fs); // forward fpu e3d to fp a
fwdfb = e3w & (e3n == ft); // forward fpu e3d to fp b
```

式中的 e3w 和 e3n 分别是 E3 级的浮点寄存器的写使能信号和浮点目的寄存器号。这两个信号是从 ID 级传过来的，它们分别对应 ID 级的 fwfpr 和 fd。

由于浮点指令的执行需要 3 级 (E1、E2 和 E3)，而上例中的两条数据相关的指令之间又隔了两条指令，使得刚好通过数据的内部前推就可解决数据相关而不用暂停流水线。两条数据相关的浮点指令之间只相隔一条指令的情况如图 10.9 所示。

注意我们不能把 E2 级的数据前推，因为它不是浮点指令的最终结果，因此我们必须要把流水线暂停一个周期，等待最终结果的出现。如何产生流水线的暂停信号，我们稍后给出，因为还有更严重的数据相关的情况，即，相邻的两条浮点指令数据相关，如图 10.10 所示。这时的流水线要暂停两个周期。

综合图 10.9 和图 10.10 的两种情况，我们给出由于浮点运算指令的数据相关而需要暂停流水线的信号 stall_fp (也要参照图 10.8):

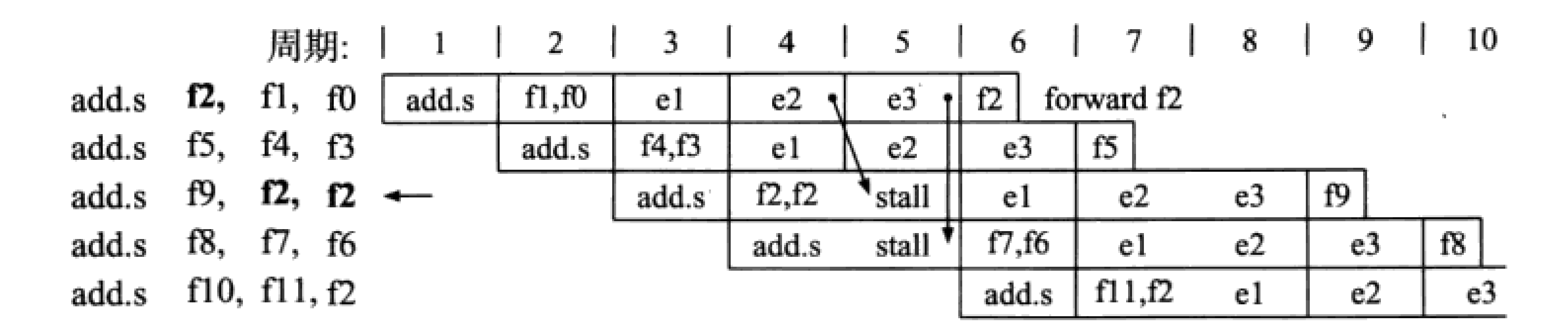


图 10.9　浮点数据的内部前推、流水线停一个周期

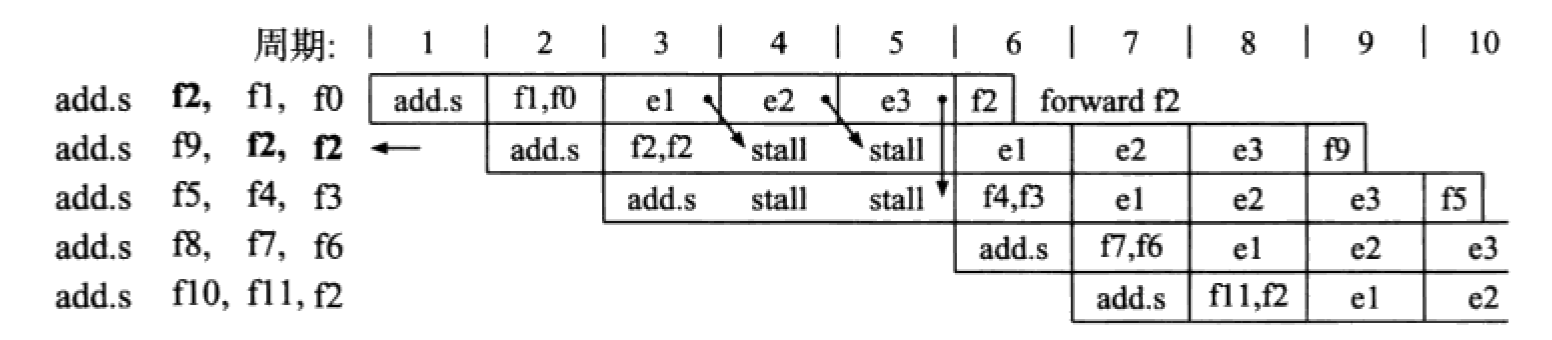


图 10.10　浮点数据的内部前推、流水线停两个周期

```
i_fs = i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt; // use fs
i_ft = i_fadd | i_fsub | i_fmul | i_fdiv;           // use ft
stall_fp = (e1w & (i_fs & (e1n == fs) | i_ft & (e1n == ft))) |
           (e2w & (i_fs & (e2n == fs) | i_ft & (e2n == ft)));
```

其中，i_fs 和 i_ft 分别表示使用 fs 寄存器操作数和 ft 寄存器操作数的浮点指令：所有的 5 条浮点运算指令都使用 fs；除了浮点开方指令，其他的 4 条指令也都使用 ft 寄存器操作数。变量 i_fadd、i_fsub、i_fmul、i_fdiv 和 i_fsqrt 是根据指令中的 op 和 func 分别对 add.s、sub.s、mul.s、div.s 和 sqrt.s 指令的译码。当 stall_fp 为 1 时，禁止在时钟的上升沿处修改程序计数器 PC 和指令寄存器 IR。因为还有其他情况也要暂停流水线，我们稍后给出 PC 和 IR 的写使能信号 wpcir。

10.3.2　lwc1 和 swc1 造成的流水线暂停及内部前推

与整数部件的 lw 和 sw 指令类似，lwc1 和 swc1 两条指令实现浮点寄存器和存储器之间的数据传送：lwc1 从存储器读出数据并将数据写入浮点寄存器堆；swc1 指令把浮点寄存器堆中的内容写入存储器。存储器地址的计算与 lw 和 sw 指令的存储器地址的计算相同。lwc1 的执行状况与 lw 基本相同，二者均可引起流水线的暂停；swc1 与 sw 稍有不同：sw 指令不会暂停流水线，而 swc1 指令即使使用内部前推技术，也有可能必须要暂停流水线。这是因为 swc1 要存储的数据是由浮点指令计算出的，而浮点运算至少要用 3 个周期 (整数运算只用一个周期)。

参照 10.1 节对 lwc1 和 swc1 的描述，我们有如图 10.11 所示的执行这两条指令所需的最基本的数据路径。lwc1 指令的流水线为 IF、ID、EXE、MEM、WB，目的寄存器号为 ft。在 ID 级，控制部件产生浮点寄存器堆的写使能信号 wfpr。ALU 在 EXE

级计算存储器地址，存储器数据在 MEM 级读出，WB 级写入浮点寄存器堆。swc1 指令的流水线为 IF、ID、EXE、MEM。在 ID 级，控制部件产生 swfp 信号，它被用来选择浮点寄存器的数据。经过流水线寄存器的锁存，被选中的数据在 MEM 级写入存储器。swc1 指令也令 wmem = 1。

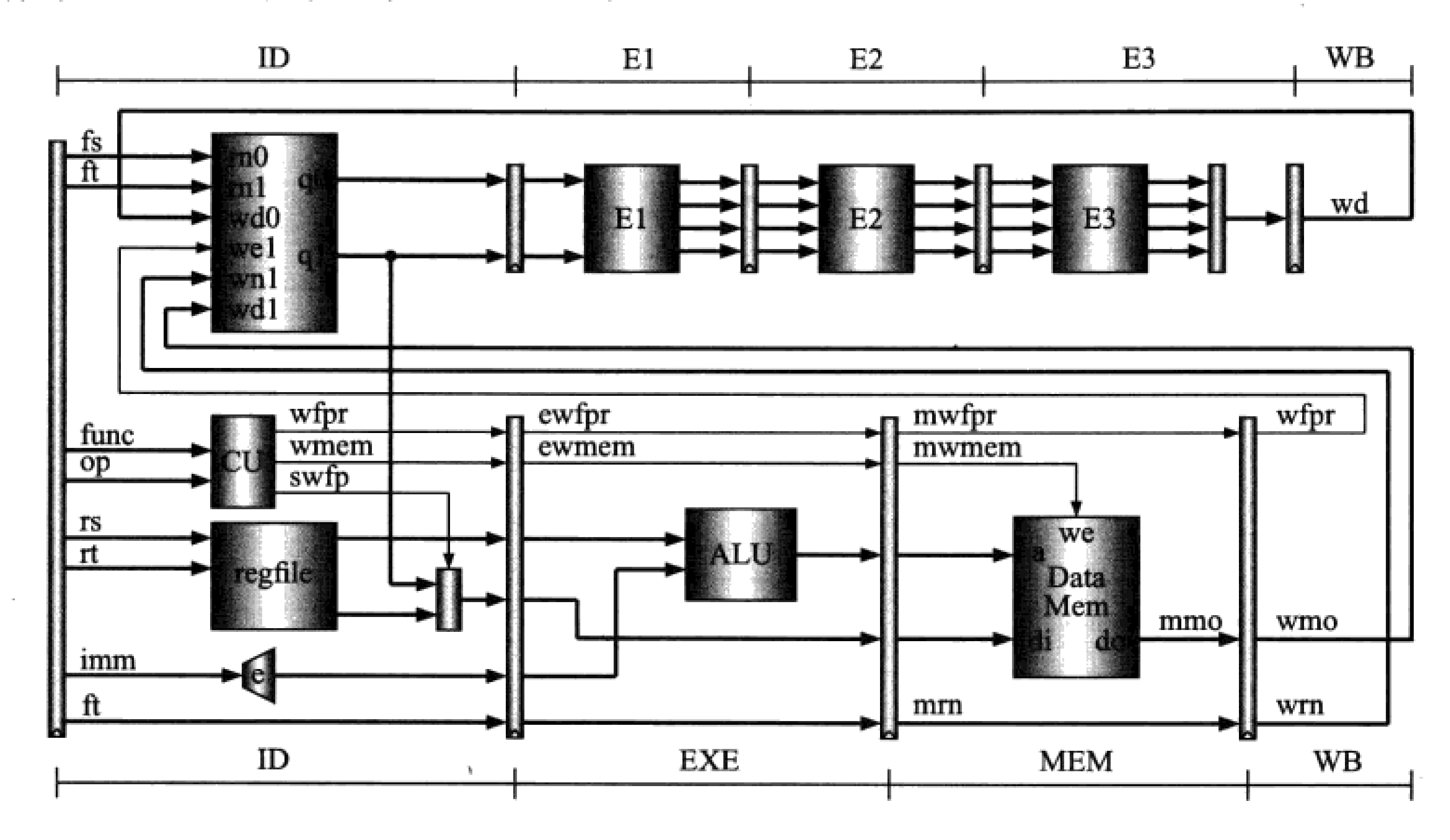


图 10.11　执行 lwc1 和 swc1 指令的基本数据路径

首先我们考虑图 10.12 执行 lwc1 的情况。第 4 条指令 add.s f2, f1, f0 使用 f1 寄存器的内容作为一个源操作数，与第 2 条指令 lwc1 f1, 100(r2) 数据相关。我们可以在第 5 个周期把刚刚从存储器读出的数据送到 add.s 指令的 ID 级。

周期:	1	2	3	4	5	6	7	8	9	10
lwc1 f0, 100(r1)	lwc1	r1	addr	mem	f0					
lwc1 **f1**, 100(r2)		lwc1	r2	addr	mem	f1	forward f1 (mmo)			
sub.s f3, f5, f4			sub.s	f5,f4	e1	e2	e3	f3		
add.s f2, **f1**, f0 ←				add.s	f1,f0	e1	e2	e3	f2	
lwc1 f4 100(r3)					lwc1	r3	addr	mem	f4	

图 10.12　与 lwc1 指令的数据相关 (内部前推)

为此，我们使用两个二选一多路器来选择存储器数据，见图 10.13。在 ID 级，当浮点运算指令的源寄存器号与 MEM 级的 lwc1 指令的目的寄存器号相同时，选择存储器数据。两个多路器的选择端 fwdla 和 fwdlb 的逻辑表达式如下。

```
wfpr  = i_lwc1 & wpcir; // fp regfile write enable
fwdla = mwfpr & (mrn == fs); // forward mmo to fp a
fwdlb = mwfpr & (mrn == ft); // forward mmo to fp b
```

wfpr 是浮点寄存器堆的写使能信号；wpcir 是 PC 和 IR 的写使能信号 (稍后给出)，和 i_lwc1 相与是为了防止流水线暂停时 lwc1 多次写浮点寄存器堆；mwfpr 是 MEM 级的 wfpr；mrn 是 MEM 级的 lwc1 指令的目的寄存器号。图 10.13 示出了存储器数据的内部前推电路，其中 mmo 是数据存储器在 MEM 级的输出。

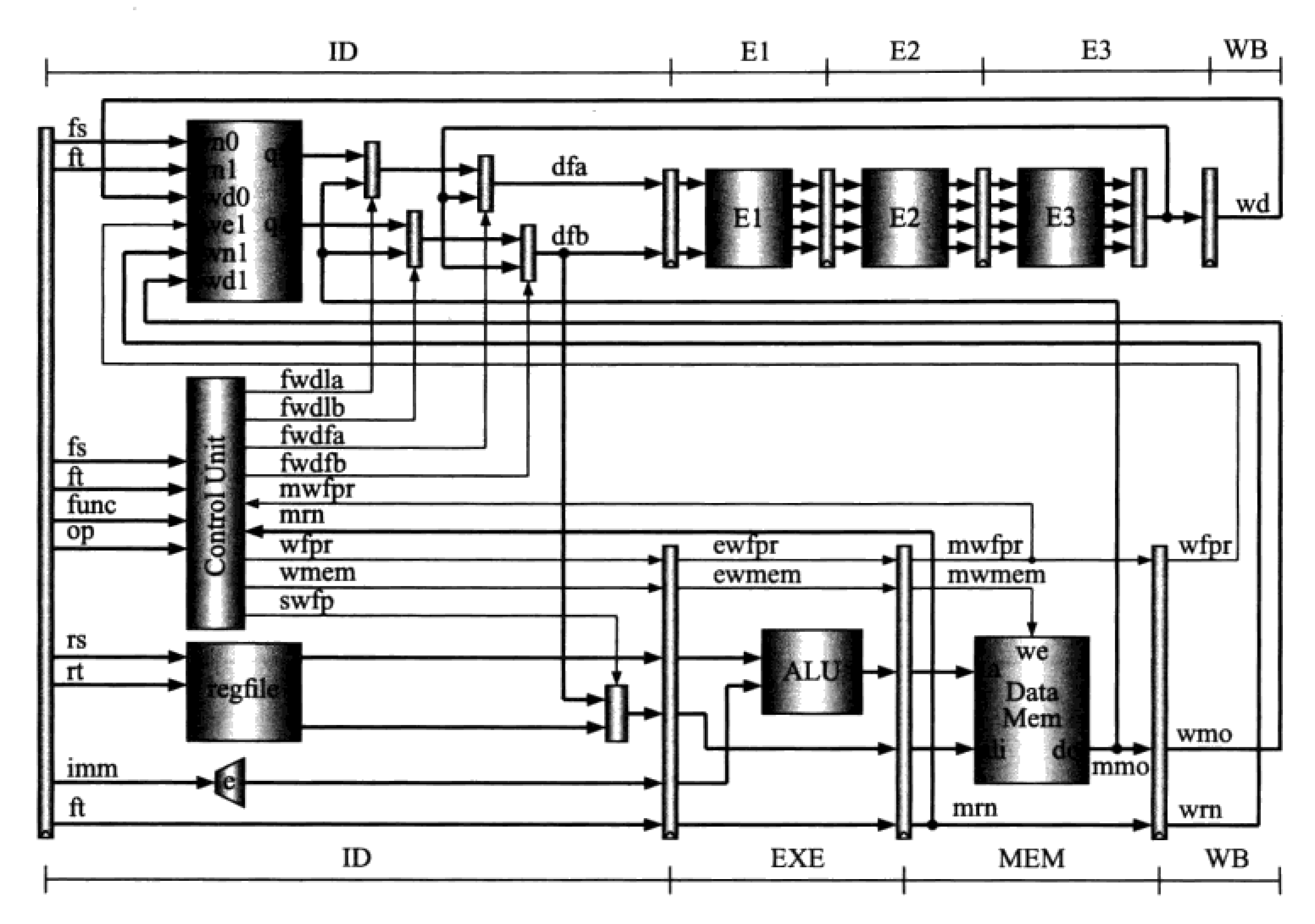


图 10.13 存储器数据的内部前推

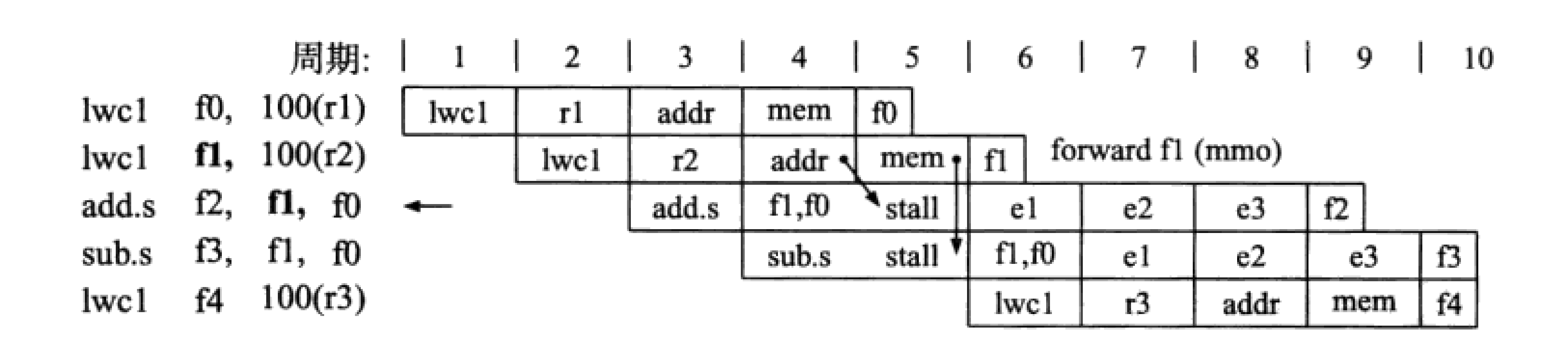


图 10.14 与 lwc1 指令的数据相关 (暂停流水线)

如果一条浮点运算指令马上使用 lwc1 取来的数据，除了要进行数据的内部前推，还需要把流水线暂停一个周期 (与 lw 指令类似)。图 10.14 所示的就是这种情况。与 lwc1 数据相关而需要暂停流水线的条件为：

```
stall_lwc1 = ewfpr & (i_fs & (ern == fs) | i_ft & (ern == ft));
```

其中，i_fs 和 i_ft 分别表示使用 fs 寄存器操作数和 ft 寄存器操作数的浮点指令，我们

已经给出了它们的逻辑表达式；ewfpr 是 EXE 级的 wfpr；ern 是 EXE 级的 lwc1 指令的目的寄存器号。

讨论完了 lwc1 的数据相关及暂停流水线的问题，现在该轮到 swc1 了。同样，我们从数据相关的最简单的情况开始。见图 10.15，第 4 条指令 swc1 f2, 100(r1) 要把第 1 条指令 add.s f2, f1, f0 的浮点加法结果保存到存储器中。swc1 指令肯定不能使用从 f2 寄存器中读出的内容，因为当 swc1 读 f2 时，add.s 还没有把加法结果写进去。

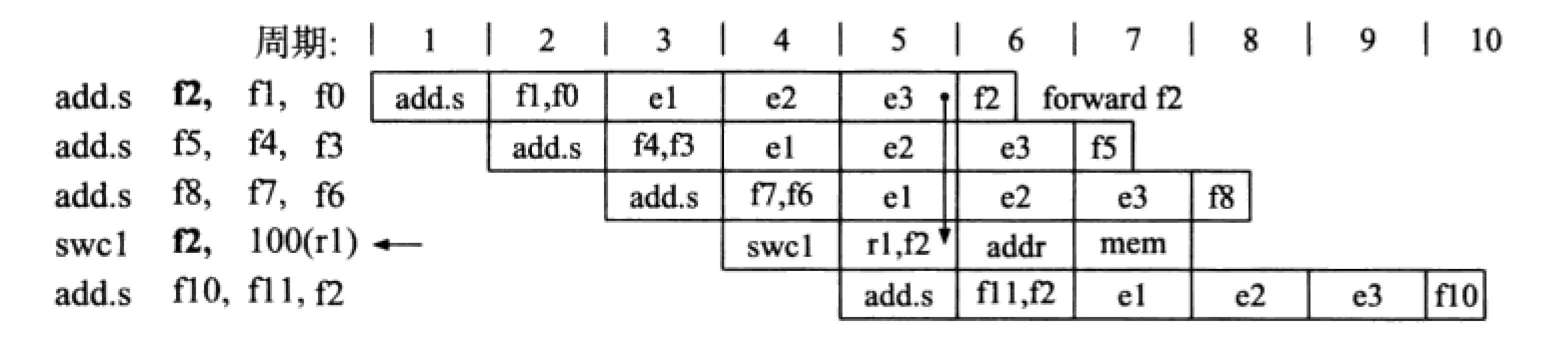


图 10.15 swc1 指令的数据相关 (前推到 ID 级)

从图中可以看出，浮点加法结果可以在第 5 个周期被前推到 swc1 的 ID 级。因此我们在 ID 级加一个二选一多路器。多路器的选择信号为 fwdf。当它为 1'时，选择浮点运算指令在 E3 级的结果 (e3d)。等讨论完了图 10.16 的情况，我们一并给出多路器选择信号的逻辑表达式。

周期:	1	2	3	4	5	6	7	8	9	10
add.s f2, f1, f0	add.s	f1,f0	e1	e2	e3	f2	forward f2			
add.s f5, f4, f3		add.s	f4,f3	e1	e2	e3	f5			
swc1 f2, 100(r1) ←			swc1	r1,f2	addr	mem				
add.s f8, f7, f6				add.s	f7,f6	e1	e2	e3	f8	
add.s f10, f11, f2					add.s	f11,f2	e1	e2	e3	f10

图 10.16 swc1 指令的数据相关 (前推到 EXE 级)

在图 10.16 中，第 3 条指令要把第 1 条指令的结果保存到存储器中。我们可以把 e3d 送到 swc1 的 EXE 级。因此我们在 EXE 级再加一个二选一多路器。注意，多路器的选择信号 fwdfe 在 ID 级生成，经过流水线寄存器送到 EXE 级，名称变为 efwdfe。当它为 1 时，选择浮点运算指令在 E3 级的结果 (e3d)。以下是 fwdf 和 fwdfe 的逻辑表达式。

```
swfp  = i_swc1; // select signal, need not & wpcir
fwdf  = swfp & e3w & (ft == e3n); // forward to id  stage
fwdfe = swfp & e2w & (ft == e2n); // forward to exe stage
```

图 10.17 所示的情况必须要把流水线暂停一个周期。由 swc1 数据相关引起的流水线暂停信号 stall_swc1 的逻辑表达式为：

```
stall_swc1 = swfp & e1w & (ft == e1n); // stall
```

式中的 swfp 是 ID 级的信号，表示 ID 级的指令是 swc1， e1w 是 E1 级的浮点运算指令写浮点寄存器堆的使能信号， e1n 是 E1 级的浮点目的寄存器号， ft 是 ID 级的 swc1 指令中的浮点源寄存器号。

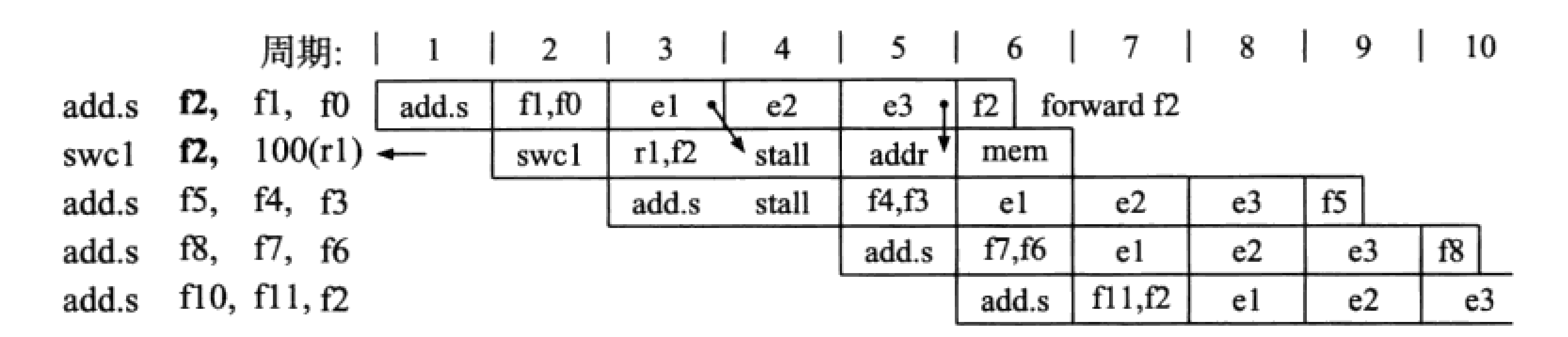


图 10.17 swc1 指令的数据相关 (暂停流水线)

综合以上的讨论，我们可以得到图 10.18 所示的 lwc1 和 swc1 指令的数据内部前推的电路结构，主要部分是几个多路器以及它们的选择信号。

图 10.18 lwc1 和 swc1 指令的数据内部前推

10.3.3 浮点除法和开方指令造成的流水线暂停

前面已经讲过，通过暂停流水线的方法，我们把浮点除法和开方指令的译码周期和 Newton-Raphson 迭代操作看成流水线的“一级”(ID 级)。对于浮点除法指令来讲，这一级有 17 个时钟周期：译码周期一个，计算迭代的初始值 x_0 一个，三次迭代 15 个 (每次 5 个)。浮点开方指令有 23 个 (三次迭代每次用 7 个周期)。

如果我们用 stall_div_sqrt 来表示浮点除法和开方指令的流水线暂停的信号，则总共有 5 个信号暂停流水线，它们是：

1) stall_lw: 整数运算指令与 lw 指令数据相关引起的流水线暂停；
2) stall_lwc1: 浮点运算指令与 lwc1 指令数据相关引起的流水线暂停；
3) stall_swc1: swc1 指令与浮点运算指令数据相关引起的流水线暂停；
4) stall_fp: 浮点运算指令之间的数据相关引起的流水线暂停；
5) stall_div_sqrt: 浮点除法和开方指令等待 Newton-Raphson 迭代操作完成。

前 4 个信号有一个共同的特点：它们只是禁止程序计数器 PC 和指令寄存器 IR 的写入，而不禁止流水线寄存器的写入 (当然要防止同一条指令执行多次)。stall_div_sqrt 除了封锁 PC 和 IR，也要禁止浮点部件的流水线寄存器的写入，见图 10.19。

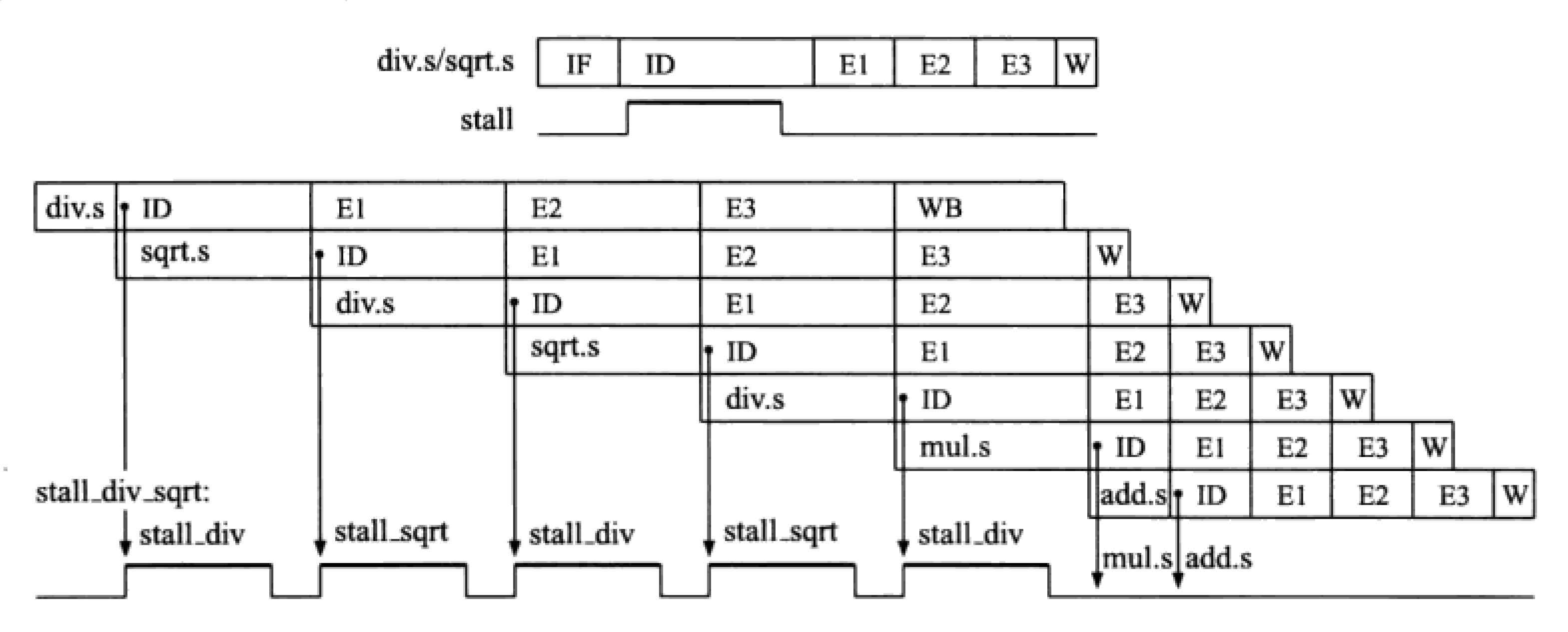


图 10.19 浮点除法和开方指令暂停流水线

在图 10.19 中，前 5 条指令是浮点除法或开方指令。在它们的 ID 级，产生相应的流水线暂停信号 stall_div 或 stall_sqrt。在流水线暂停期间，浮点部件完成 Newton-Raphson 迭代。因为我们把这个特殊的 ID 看成流水线的一级，所以它有多长，在此期间的浮点指令的 E1、E2、E3 和 WB 周期就会有多长。因此，浮点部件中的所有流水线寄存器的写使能信号 w_fp_pipe_reg 为：

```
stall_div_sqrt =  stall_div | stall_sqrt;
w_fp_pipe_reg  = ~stall_div_sqrt; // fp pipe_reg write enable
```

当以上 5 个流水线暂停信号中的任何一个为 1 时，都要暂停流水线。因此，PC 和 IR 的写使能信号 wpcir 为：

```
wpcir = ~(stall_div_sqrt | stall_others);
stall_others = stall_lw | stall_fp | stall_lwc1 | stall_swc1;
```

为什么要单独设一个 stall_others 信号？因为它与 stall_div_sqrt 不同。我们知道，暂停流水线只是禁止修改 PC 和 IR，而已经在 ID 级的指令还是会沿着流水线“流”下去。这就会导致一条指令重复执行两次或两次以上。为了避免由此而错误地修改 CPU 的状态，我们在 ID 级把所有的写信号在流水线暂停期间都设置为无效：

```
wreg = (i_add  | i_sub | i_and  | i_or   | i_xor | i_sll  |
        i_srl  | i_sra | i_addi | i_andi | i_ori | i_xori |
        i_lw   | i_lui | i_jal) & wpcir; // regfile write ena.
wmem = (i_sw   | i_swc1) & wpcir; // memory write
wfpr =  i_lwc1 & wpcir; // fp regfile write ena. (lwc1)
wf   =  i_fs   & wpcir; // fp regfile write ena. (fp operation)
i_fs =  i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt; // use fs
// fop:  000: fadd
//       001: fsub
//       01x: fmul
//       10x: fdiv
//       11x: fsqrt
fop[0] = i_fsub; // fpu operation control code
fop[1] = i_fmul | i_fsqrt;
fop[2] = i_fdiv | i_fsqrt;
fc = fop & {3{~stall_others}};
```

前 4 个写信号均与 (&) 上了 wpcir (封锁写信号)。最后一个信号 fc 是浮点操作的控制码。根据 fc，我们可以得到浮点除法指令和浮点开方指令的启动执行信号 fdiv 和 fsqrt。而 fdiv 和 fsqrt 会启动各自的计数器以完成 Newton-Raphson 迭代并且产生各自的暂停流水线的信号：

```
fdiv  = fc[2] & ~fc[1]; // start the execution of div.s
fsqrt = fc[2] &  fc[1]; // start the execution of sqrt.s
stall_div  = fdiv  & (count == 0) | busy;
stall_sqrt = fsqrt & (count == 0) | busy;
```

为什么 fc 的产生要与上 stall_others 的“非”而不是直接与上 wpcir？如果在 fc 的逻辑表达式中直接与上 {3{wpcir}}，会造成组合逻辑的“循环”：不知道信号的源头在哪里，因为产生 wpcir 时是把 fc 作为输入信号使用的。那么，因为 stall_others 信号中不包含 stall_div_sqrt，会不会造成一条浮点指令被执行多次？答案是不会，因为 stall_div_sqrt 封锁了浮点部件的流水线寄存器，指令“流”不下去。

10.4 带有 FPU 的流水线 CPU 的总体结构及 Verilog HDL 代码

有了以上各节的描述，现在我们正式给出带有 FPU 的流水线 CPU 的具体电路以及 Verilog HDL 代码。

10.4.1 带有 FPU 的流水线 CPU 的具体电路

图 10.20 是 CPU 的总体电路图。它由整数部件模块 iu、浮点部件模块 fpu、两个写端口的浮点寄存器堆 regfile2w 和 4 个二选一多路器组成。

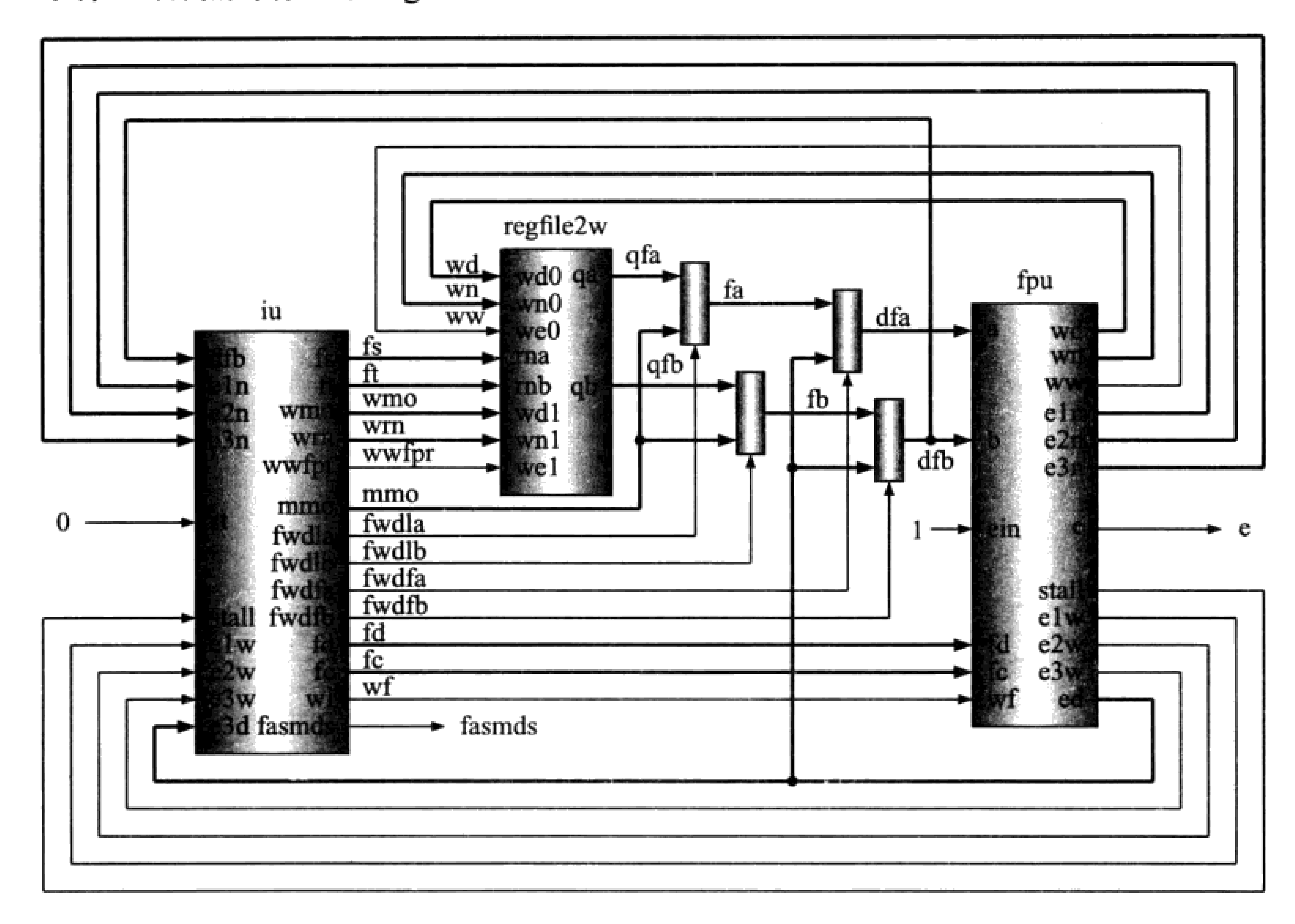


图 10.20 带有浮点部件的流水线 CPU

浮点部件在 WB 级的运行结果 wd 经 0 号端口写入浮点寄存器堆中的 wn 寄存器，ww 是写使能信号；lwc1 指令取来的存储器数据 wmo (WB 级) 经 1 号端口写入 wrn 寄存器，写使能信号为 wwfpr。浮点寄存器堆两个读端口的输出为 qfa 和 qfb，它们分别是 fs 寄存器和 ft 寄存器的内容。

左边的两个多路器用来前推 MEM 级的存储器数据 mmo，其选择信号分别为 fwdla 和 fwdlb；右边的两个多路器用来前推浮点部件 E3 级的计算结果 e3d (fpu 的 ed)，其选择信号分别为 fwdfa 和 fwdfb。最右边多路器的输出 dfb 是 swc1 指令要往数据存储器中写入的数据，所以要把它送给 iu。其他的直接连接 iu 和 fpu 的信号用于浮点操作的控制和数据相关的判断。

图中有两个输出信号没有使用，它们是：(1) 指出当前在 ID 级的指令是浮点加减乘除或开方指令的 fasmds 信号；(2) 流水线寄存器的写使能信号 e。两个输入信号 st 和 ein 分别接 0 和 1。这些信号将在第 11 章给出的多线程 CPU 的设计中用到。以下两小节分别详细介绍整数部件模块 iu 和浮点部件模块 fpu 的内部电路。

10.4.2　浮点部件 FPU 的具体电路

浮点部件执行浮点运算指令。图 10.21 示出图 10.20 中 fpu 模块的详细电路。

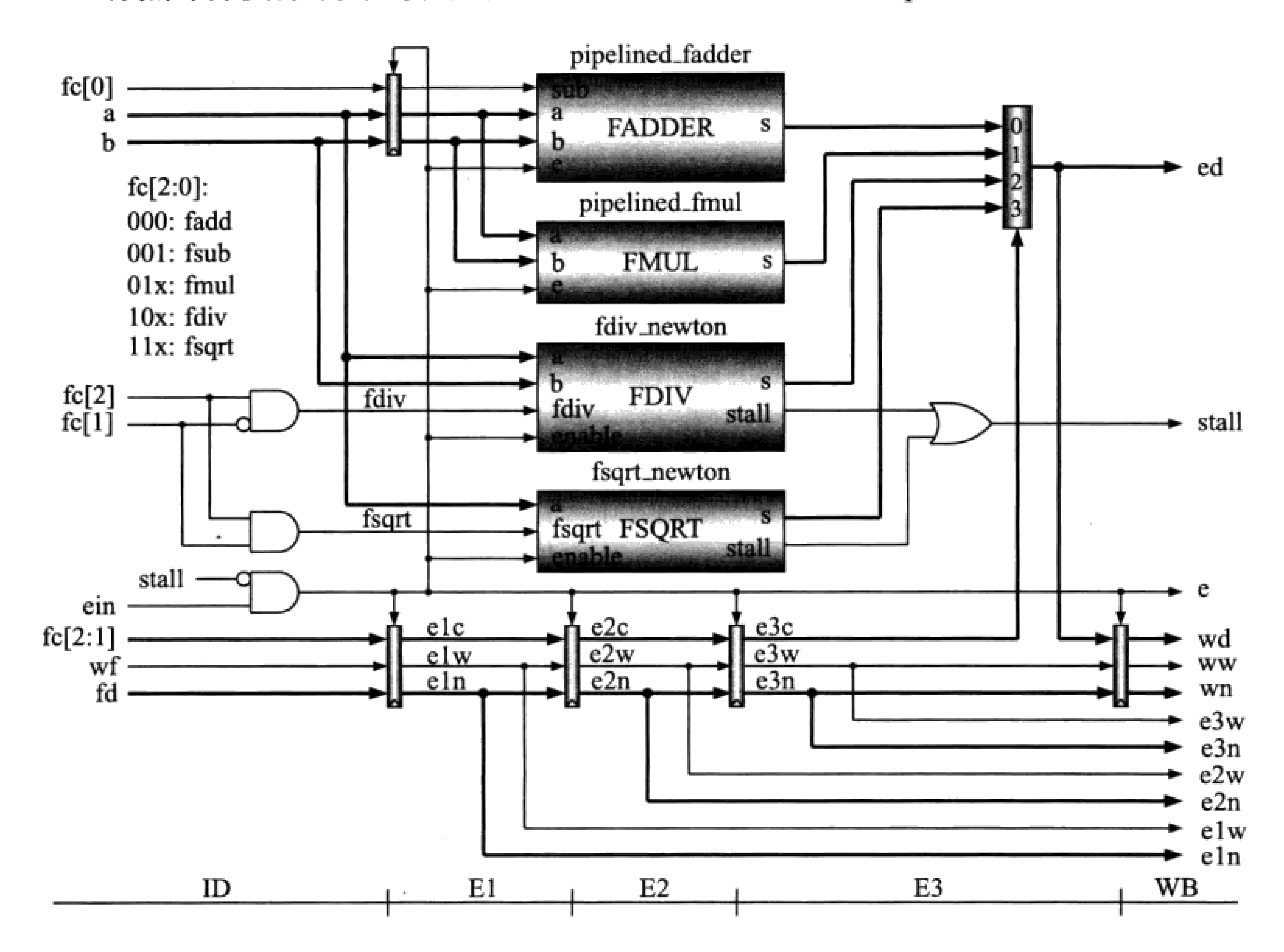


图 10.21　浮点部件 FPU

输入信号有：(1) 两个 32 位的单精度浮点数 a 和 b；(2) 浮点操作控制码 fc (3 位)，其编码的意义已在图中给出。fc 的最高两位用作四选一多路器的选择信号，最低位控制是加还是减；(3) 浮点运算指令的浮点寄存器堆写使能信号 wf；(4) 浮点运算指令的目的寄存器号 fd。后三个信号 fc、wf 和 fd 来自 iu 模块的控制部件。

输出信号有：(1) E3 级的运算结果 ed，用于内部前推；(2) 浮点除法或开方指令引起的流水线暂停信号 stall；(3) WB 级的运算结果 wd；(4) WB 级的浮点寄存器堆写使能信号 ww；(5) WB 级的浮点运算指令的目的寄存器号 wn；(6) E1、E2 和 E3 级的寄存器堆的写使能和目的寄存器号 e1w、e1n、e2w、e2n、e3w 和 e3n。这些信号被控制部件用来判断数据是否相关以产生多路器的选择信号和流水线的暂停信号。

该模块画有 5 个流水线寄存器。其他的流水线寄存器藏在 4 个浮点运算模块内部。这 4 个模块分别完成浮点加减、乘、除和开方。我们可以根据每个模块上方的名称，在第 9 章中查到它们的 Verilog HDL 代码。在本章的 CPU 实现中，我们忽略了浮点舍入方式的选择，将其设置为固定的就近舍入。

10.4.3 整数部件 IU 的具体电路

图 10.22 示出的是图 10.20 中 iu 模块的详细电路。与第 8 章的流水线 CPU 相比，整数部件新加的主要功能是对浮点指令译码并提供相应的控制信号。例如为浮点运算指令提供操作控制码 fc、浮点目的寄存器号 fd 和相应的浮点寄存器堆写信号 wf；为 lwc1 指令提供浮点寄存器堆的写信号 wfpr 和相应的数据 wmo；为 swc1 指令提供浮点寄存器数据的选择信号 swfp 和相应的数据 dfb。

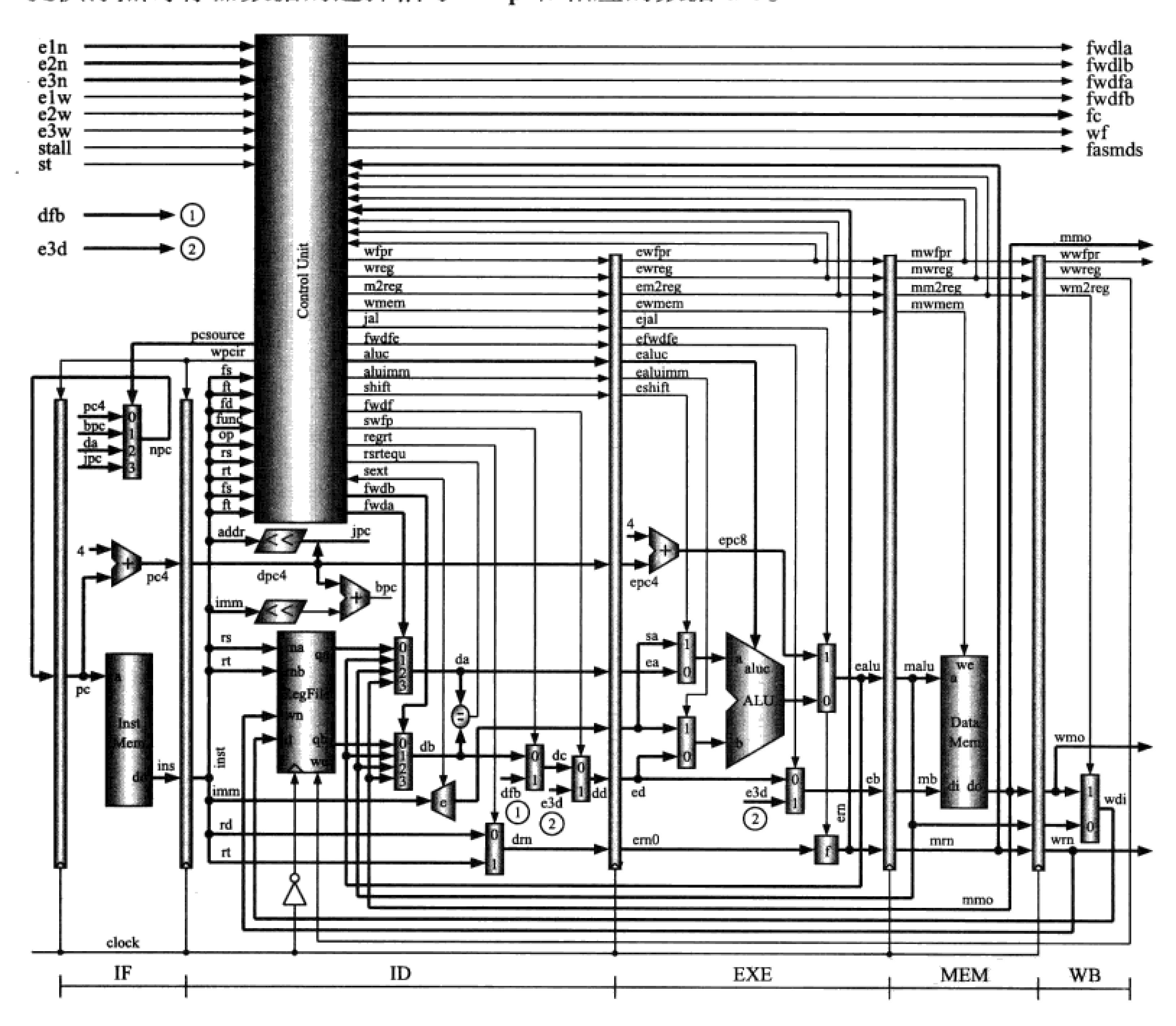


图 10.22 整数部件 IU

与数据内部前推有关的信号有：执行 lwc1 指令时的存储器数据前推信号 fwdla 和 fwdlb 以及相应的数据 mmo；执行 swc1 指令时的浮点寄存器数据前推信号 fwdf (ID 级前推) 和 fwdfe (EXE 级前推) 以及相应的数据 e3d；执行浮点运算指令时的浮点结果 (e3d) 前推信号 fwdfa 和 fwdfb。生成这些信号时，控制部件需要知道 fpu 的内部状态：e1w、e1n、e2w、e2n、e3w 和 e3n。最后还有一个来自 fpu 的 stall 信号，表示浮点部件正在执行浮点除法或开方指令，用于暂停流水线。

10.4.4 带有 FPU 的流水线 CPU 的 Verilog HDL 代码

本小节列出 CPU 的 Verilog HDL 代码。首先从最顶层的代码开始。读者可以对照图 10.20 阅读以下的代码，但要注意模块的输入输出信号要比图中多。多出的信号有些是必需的而图中没有画，比如时钟信号 clock；有些则是为使逻辑测试方便而设置的，比如计数器 count_div 和 count_sqrt。

```
module fpu_1_iu (clock,memclock,resetn,pc,inst,ealu,malu,walu,wn,
                 wd,ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,
                 stall,count_div,count_sqrt,e1n,e2n,e3n,e3d,e);
    input clock,memclock,resetn;
    output [31:0] pc,inst,ealu,malu,walu;
    output [31:0] e3d,wd;
    output [4:0] e1n,e2n,e3n,wn;
    output ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,stall;
    output e; // for multithreading CPU, not used here
    output [4:0] count_div,count_sqrt; // for testing
```

调用整数部件 iu 模块：

```
    wire [31:0] qfa,qfb,fa,fb,dfa,dfb,mmo,wmo; // for iu
    wire [4:0]  fs,ft,fd;
    wire [2:0]  fc;
    wire        fwdla,fwdlb,fwdfa,fwdfb,wf,fasmds;
    wire        e1w,e2w,e3w,wwfpr;
    iu i_u (e1n,e2n,e3n, e1w,e2w,e3w, stall,1'b0, // st = 0
            dfb,e3d, clock,memclock,resetn,
            fs,ft,wmo,wrn,wwfpr,mmo,fwdla,fwdlb,fwdfa,fwdfb,fd,fc,wf,fasmds,
            pc,inst,ealu,malu,walu, // for testing
            stall_lw,stall_fp,stall_lwc1,stall_swc1); // for testing
```

浮点寄存器堆和 4 个二选一多路器：

```
    wire [4:0] wrn;
    regfile2w fpr (fs,ft,wd,wn,ww,wmo,wrn,wwfpr,~clock,resetn,qfa,qfb);
    mux2x32 fwd_f_load_a (qfa,mmo,fwdla,fa); // forward lwc1 to fp a
    mux2x32 fwd_f_load_b (qfb,mmo,fwdlb,fb); // forward lwc1 to fp b
    mux2x32 fwd_f_res_a  (fa,e3d,fwdfa,dfa); // forward fp res to fp a
    mux2x32 fwd_f_res_b  (fb,e3d,fwdfb,dfb); // forward fp res to fp b
```

调用浮点部件 fpu 模块：

```
    wire [1:0] e1c,e2c,e3c; // for fpu
    fpu fp_unit (dfa,dfb,fc,wf,fd,1'b1,clock,resetn,e3d,wd,wn,ww,
                 stall,e1n,e1w,e2n,e2w,e3n,e3w,
                 e1c,e2c,e3c,count_div,count_sqrt,e);
endmodule
```

以下是浮点部件 fpu 模块 (对照图 10.21)：

```
module fpu (a,b,fc,wf,fd,ein,clk,clrn,ed,wd,wn,ww,stall,
            e1n,e1w,e2n,e2w,e3n,e3w, e1c,e2c,e3c,count_div,
            count_sqrt,e);
    input  [31:0] a,b; // 32-bit fp numbers
    input   [2:0] fc;  // 000:add 001:sub 01x:mul 10x:div 11x:sqrt
    input         wf;  // write fp regfile
    input   [4:0] fd;  // fp destination reg number
    input         ein; // enable input
    input         clk,clrn;
    output [31:0] ed,wd; // wd: fp result
    output        e1w,e2w,e3w,ww; // write fp regfile
    output  [4:0] e1n,e2n,e3n,wn; // reg numbers
    output        stall,e; // caused by fdiv and fsqrt
    output  [1:0] e1c,e2c,e3c; // for testing
    output  [4:0] count_div,count_sqrt; // for testing
    reg    [31:0] wd;
    reg           sub;
    reg    [31:0] efa,efb;
    wire   [31:0] s_add,s_mul,s_div,s_sqrt;
    reg     [1:0] e1c,e2c,e3c;
    reg           e1w,e2w,e3w,ww;
    reg     [4:0] e1n,e2n,e3n,wn;
    wire          busy_div,stall_div,busy_sqrt,stall_sqrt;
    wire   [25:0] reg_x_div,reg_x_sqrt;
    wire          fdiv  = fc[2] & ~fc[1];
    wire          fsqrt = fc[2] &  fc[1];
```

调用第 9 章给出的 4 个浮点运算模块，舍入方式 rm = 00：

```
    pipelined_fadder f_add (efa,efb,sub,2'b0,s_add,clk,clrn,e);
    pipelined_fmul   f_mul     (efa,efb,2'b0,s_mul,clk,clrn,e);
    fdiv_newton f_div (a,b,2'b0,fdiv, e,clk,clrn,s_div, busy_div,
                       stall_div, count_div, reg_x_div);
    fsqrt_newton f_sqrt (a,2'b0,fsqrt,e,clk,clrn,s_sqrt,busy_sqrt,
                       stall_sqrt,count_sqrt,reg_x_sqrt);
```

产生流水线暂停信号和流水线寄存器的写使能、选择运算结果：

```
    assign stall = stall_div | stall_sqrt;
    assign e = ~stall & ein;
    mux4x32 fsel (s_add,s_mul,s_div,s_sqrt,e3c,ed);
```

流水线寄存器，受写使能信号控制：

```
    always @ (negedge clrn or posedge clk)
      if (clrn == 0) begin // pipeline registers
          sub <= 0;          efa <= 0;          efb <= 0;
```

```
            e1c <= 0;           e1w <= 0;           e1n <= 0;
            e2c <= 0;           e2w <= 0;           e2n <= 0;
            e3c <= 0;           e3w <= 0;           e3n <= 0;
            wd  <= 0;           ww  <= 0;           wn  <= 0;
        end else if (e) begin
            sub <= fc[0];       efa <= a;           efb <= b;
            e1c <= fc[2:1];     e1w <= wf;          e1n <= fd;
            e2c <= e1c;         e2w <= e1w;         e2n <= e1n;
            e3c <= e2c;         e3w <= e2w;         e3n <= e2n;
            wd  <= ed;          ww  <= e3w;         wn  <= e3n;
        end
endmodule
```

以下是整数部件 iu 模块 (对照图 10.22):

```
module iu (e1n,e2n,e3n, e1w,e2w,e3w, stall,st,
           dfb,e3d, clock,memclock,resetn,
           fs,ft,wmo,wrn,wwfpr,mmo,fwdla,fwdlb,fwdfa,fwdfb,fd,fc,wf,fasmds,
           pc,inst,ealu,malu,walu, // for testing
           stall_lw,stall_fp,stall_lwc1,stall_swc1); // for testing
    input [31:0] dfb,e3d;
    input [4:0] e1n,e2n,e3n;
    input       e1w,e2w,e3w, stall,st, clock,memclock,resetn;
    output [31:0] pc,inst,ealu,malu,walu;
    output [31:0] mmo,wmo;
    output [4:0]  fs,ft,fd,wrn;
    output [2:0]  fc;
    output        wwfpr,fwdla,fwdlb,fwdfa,fwdfb,wf,fasmds;
    output        stall_lw,stall_fp,stall_lwc1,stall_swc1;
    wire [31:0] bpc,jpc,npc,pc4,ins,dpc4,inst,qa,qb,da,db,dimm,dc,dd;
    wire [31:0] simm,epc8,alua,alub,ealu0,ealu,sa,eb,mmo,wdi;
    wire [5:0]  op,func;
    wire [4:0]  rs,rt,rd,fs,ft,fd,drn,ern;
    wire [3:0]  aluc;
    wire [1:0]  pcsource,fwda,fwdb;
    wire        wpcir;
    wire        wreg,m2reg,wmem,aluimm,shift,jal;
    wire [31:0] qfa,qfb,fa,fb,dfa,dfb,efb,e3d;
    wire [4:0]  e1n,e2n,e3n,wn;
    wire [2:0]  fc;
    wire [1:0]  e1c,e2c,e3c;
    reg ewfpr,ewreg,em2reg,ewmem,ejal,efwdfe,ealuimm,eshift;
    reg mwfpr,mwreg,mm2reg,mwmem;
    reg wwfpr,wwreg,wm2reg;
    reg [31:0] epc4,ea,ed,eimm,malu,mb,wmo,walu;
    reg [4:0] ern0,mrn,wrn;
    reg [3:0] ealuc;
```

IF 级：

```
dffe32 program_counter (npc,clock,resetn,wpcir,pc); // PC
cla32 pc_plus4 (pc,32'h4,1'b0,pc4); // PC+4
mux4x32 next_pc (pc4,bpc,da,jpc,pcsource,npc); // Next PC
inst_mem i_mem (pc,ins); // instruction memory
```

IF 级与 ID 级之间的流水线寄存器：

```
dffe32 pc_4_r (pc4,clock,resetn,wpcir,dpc4); // PC+4 reg
dffe32 inst_r (ins,clock,resetn,wpcir,inst); // IR
```

ID 级：

```
assign op   = inst[31:26];
assign rs   = inst[25:21];
assign rt   = inst[20:16];
assign rd   = inst[15:11];
assign func = inst[5:0];
assign simm = {{16{sext&inst[15]}},inst[15:0]};
assign jpc  = {dpc4[31:28],inst[25:0],2'b00}; // jump target
cla32 br_addr (dpc4,{simm[29:0],2'b00},1'b0,bpc); // branch target
regfile rf (rs,rt,wdi,wrn,wwreg,~clock,resetn,qa,qb); // reg file
mux4x32 alu_a (qa,ealu,malu,mmo,fwda,da); // forward A
mux4x32 alu_b (qb,ealu,malu,mmo,fwdb,db); // forward B
wire swfp,regrt,sext,fwdf,fwdfe,wfpr;
mux2x32 store_f (db,dfb,swfp,dc); // swc1
mux2x32 fwd_f_d (dc,e3d,fwdf,dd); // forward fp result
wire    rsrtequ = ~|(da^db); // rsrtequ = (da == db)
mux2x5 des_reg_no (rd,rt,regrt,drn); // destination reg
control cu (op,func,rs,rt,fs,ft,rsrtequ, // control unit
           ewfpr,ewreg,em2reg,ern,       // from iu
           mwfpr,mwreg,mm2reg,mrn,       // from iu
           e1w,e1n,e2w,e2n,e3w,e3n,stall,st, // from fpu
           pcsource,wpcir,wreg,m2reg,wmem,jal,aluc, // iu
           aluimm,shift,sext,regrt,fwda,fwdb, // iu
           swfp,fwdf,fwdfe,wfpr,         // used by iu
           fwdla,fwdlb,fwdfa,fwdfb,fc,wf,fasmds, // used by fpu
           stall_lw,stall_fp,stall_lwc1,stall_swc1); // for testing
// for fpu
assign ft = inst[20:16];
assign fs = inst[15:11];
assign fd = inst[10:6];
```

ID 级与 EXE 级之间的流水线寄存器：

```
always @(negedge resetn or posedge clock)
  if (resetn == 0) begin
      ewfpr   <= 0;          ewreg   <= 0;
      em2reg  <= 0;          ewmem   <= 0;
      ejal    <= 0;          ealuimm <= 0;
```

```
        efwdfe   <= 0;           ealuc    <= 0;
        eshift   <= 0;           epc4     <= 0;
        ea       <= 0;           ed       <= 0;
        eimm     <= 0;           ern0     <= 0;
    end else begin
        ewfpr    <= wfpr;        ewreg    <= wreg;
        em2reg   <= m2reg;       ewmem    <= wmem;
        ejal     <= jal;         ealuimm  <= aluimm;
        efwdfe   <= fwdfe;       ealuc    <= aluc;
        eshift   <= shift;       epc4     <= dpc4;
        ea       <= da;          ed       <= dd;
        eimm     <= simm;        ern0     <= drn;
    end
```

EXE 级：

```
cla32 ret_addr (epc4,32'h4,1'b0,epc8); // PC+8
assign sa = {eimm[5:0],eimm[31:6]}; // shift amount
mux2x32 alu_ina (ea,sa,eshift,alua); // ALU input A
mux2x32 alu_inb (eb,eimm,ealuimm,alub); // ALU input B
mux2x32 save_pc8 (ealu0,epc8,ejal,ealu); // PC+8 if jal
wire z;
alu al_unit (alua,alub,ealuc,ealu0,z); // ALU
assign ern = ern0 | {5{ejal}}; // r31 for jal
mux2x32 fwd_f_e (ed,e3d,efwdfe,eb); // forward fp result
```

EXE 级与 MEM 级之间的流水线寄存器：

```
always @(negedge resetn or posedge clock)
    if (resetn == 0) begin
        mwfpr    <= 0;           mwreg    <= 0;
        mm2reg   <= 0;           mwmem    <= 0;
        malu     <= 0;           mb       <= 0;
        mrn      <= 0;
    end else begin
        mwfpr    <= ewfpr;       mwreg    <= ewreg;
        mm2reg   <= em2reg;      mwmem    <= ewmem;
        malu     <= ealu;        mb       <= eb;
        mrn      <= ern;
    end
```

MEM 级：

```
data_mem d_mem (mwmem,malu,mb,clock,memclock,memclock,mmo);
```

MEM 级与 WB 级之间的流水线寄存器：

```
always @(negedge resetn or posedge clock)
    if (resetn == 0) begin
        wwfpr    <= 0;           wwreg    <= 0;
```

```
        wm2reg  <= 0;           wmo     <= 0;
        walu    <= 0;           wrn     <= 0;
    end else begin
        wwfpr   <= mwfpr;       wwreg   <= mwreg;
        wm2reg  <= mm2reg;      wmo     <= mmo;
        walu    <= malu;        wrn     <= mrn;
    end
```

WB 级：

```
    mux2x32 wb_sel (walu,wmo,wm2reg,wdi);
endmodule
```

以下是控制部件模块的代码，所有的控制信号均在此模块产生。

```
module control (op,func,rs,rt,fs,ft,rsrtequ,                    // control unit
                ewfpr,ewreg,em2reg,ern,                         // from iu
                mwfpr,mwreg,mm2reg,mrn,                         // from iu
                e1w,e1n,e2w,e2n,e3w,e3n,stall_div_sqrt,st,      // from fpu
                pcsource,wpcir,wreg,m2reg,wmem,jal,aluc,        // iu
                aluimm,shift,sext,regrt,fwda,fwdb,              // iu
                swfp,fwdf,fwdfe,wfpr,                           // for lwc1,swc1
                fwdla,fwdlb,fwdfa,fwdfb,fc,wf,fasmds,           // used by fpu
                stall_lw,stall_fp,stall_lwc1,stall_swc1);       // for testing
    input  rsrtequ, ewreg,em2reg,ewfpr, mwreg,mm2reg,mwfpr;
    input  e1w,e2w,e3w,stall_div_sqrt,st;
    input  [5:0] op,func;
    input  [4:0] rs,rt,fs,ft,ern,mrn,e1n,e2n,e3n;
    output wpcir,wreg,m2reg,wmem,jal,aluimm,shift,sext,regrt;
    output swfp,fwdf,fwdfe;
    output fwdla,fwdlb,fwdfa,fwdfb;
    output wfpr,wf,fasmds;
    output [1:0] pcsource,fwda,fwdb;
    output [3:0] aluc;
    output [2:0] fc;
    output stall_lw,stall_fp,stall_lwc1,stall_swc1; // for testing
```

指令译码：

```
    wire r_type,i_add,i_sub,i_and,i_or,i_xor,i_sll,i_srl,i_sra,i_jr;
    and(r_type,~op[5],~op[4],~op[3],~op[2],~op[1],~op[0]); // r format
    and(i_add,r_type, func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
    and(i_sub,r_type, func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
    and(i_and,r_type, func[5],~func[4],~func[3], func[2],~func[1],~func[0]);
    and(i_or, r_type, func[5],~func[4],~func[3], func[2],~func[1], func[0]);
    and(i_xor,r_type, func[5],~func[4],~func[3], func[2], func[1],~func[0]);
    and(i_sll,r_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
    and(i_srl,r_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
    and(i_sra,r_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
    and(i_jr, r_type,~func[5],~func[4], func[3],~func[2],~func[1],~func[0]);
```

```
wire i_addi,i_andi,i_ori,i_xori,i_lw,i_sw,i_beq,i_bne,i_lui;
and(i_addi,~op[5],~op[4], op[3],~op[2],~op[1],~op[0]);
and(i_andi,~op[5],~op[4], op[3], op[2],~op[1],~op[0]);
and(i_ori, ~op[5],~op[4], op[3], op[2],~op[1], op[0]);
and(i_xori,~op[5],~op[4], op[3], op[2], op[1],~op[0]);
and(i_lw,   op[5],~op[4],~op[3],~op[2], op[1], op[0]);
and(i_sw,   op[5],~op[4], op[3],~op[2], op[1], op[0]);
and(i_beq, ~op[5],~op[4],~op[3], op[2],~op[1],~op[0]);
and(i_bne, ~op[5],~op[4],~op[3], op[2],~op[1], op[0]);
and(i_lui, ~op[5],~op[4], op[3], op[2], op[1], op[0]);
wire i_j,i_jal;
and(i_j,   ~op[5],~op[4],~op[3],~op[2], op[1],~op[0]);
and(i_jal, ~op[5],~op[4],~op[3],~op[2], op[1], op[0]);
wire f_type,i_lwc1,i_swc1,i_fadd,i_fsub,i_fmul,i_fdiv,i_fsqrt;
and(f_type,~op[5], op[4],~op[3],~op[2],~op[1], op[0]); // f format
and(i_lwc1, op[5], op[4],~op[3],~op[2],~op[1], op[0]);
and(i_swc1, op[5], op[4], op[3],~op[2],~op[1], op[0]);
and(i_fadd,f_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_fsub,f_type,~func[5],~func[4],~func[3],~func[2],~func[1], func[0]);
and(i_fmul,f_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_fdiv,f_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_fsqrt,f_type,~func[5],~func[4],~func[3],func[2],~func[1],~func[0]);
```

与 lw 指令相关情况下的流水线暂停和内部前推：

```
wire i_rs = i_add  | i_sub | i_and  | i_or  | i_xor | i_jr  | i_addi |
            i_andi | i_ori | i_xori | i_lw  | i_sw  | i_beq | i_bne  |
            i_lwc1 | i_swc1;
wire i_rt = i_add  | i_sub | i_and  | i_or  | i_xor | i_sll | i_srl  |
            i_sra  | i_sw  | i_beq  | i_bne;
assign stall_lw = ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) |
                                                 i_rt & (ern == rt));
reg [1:0] fwda, fwdb;
always @ (ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or
          rt) begin
    fwda = 2'b00; // default forward a: no hazards
    if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
        fwda = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
            fwda = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                fwda = 2'b11; // select mem_lw
            end
        end
    end
    fwdb = 2'b00; // default forward b: no hazards
```

```
    if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
        fwdb = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
            fwdb = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                fwdb = 2'b11; // select mem_lw
            end
        end
    end
end
```

IU 的其他控制信号：

```
assign wreg = (i_add  | i_sub  | i_and  | i_or   | i_xor | i_sll  |
               i_srl  | i_sra  | i_addi | i_andi | i_ori | i_xori |
               i_lw   | i_lui  | i_jal) & wpcir;
assign regrt = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui | i_lwc1;
assign jal   = i_jal;
assign m2reg = i_lw;
assign shift = i_sll  | i_srl  | i_sra;
assign aluimm = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui | i_sw |
                i_lwc1 | i_swc1;
assign sext   = i_addi | i_lw | i_sw | i_beq | i_bne | i_lwc1 | i_swc1;
assign aluc[3] = i_sra;
assign aluc[2] = i_sub | i_or  | i_srl | i_sra | i_ori  | i_lui;
assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_beq  |
                 i_bne | i_lui;
assign aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori;
assign wmem = (i_sw | i_swc1) & wpcir;
assign pcsource[1] = i_jr | i_j | i_jal;
assign pcsource[0] = i_beq & rsrtequ | i_bne & ~rsrtequ | i_j | i_jal;
```

与 FPU 有关的控制信号：

```
// fop:  000: fadd  001: fsub  01x: fmul  10x: fdiv  11x: fsqrt
wire [2:0]  fop;
assign fop[0] = i_fsub; // fpu operation control code
assign fop[1] = i_fmul | i_fsqrt;
assign fop[2] = i_fdiv | i_fsqrt;
// stall caused by fp data harzards
wire i_fs = i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt; // use fs
wire i_ft = i_fadd | i_fsub | i_fmul | i_fdiv;           // use ft
assign stall_fp = (e1w & (i_fs & (e1n == fs) | i_ft & (e1n == ft))) |
                  (e2w & (i_fs & (e2n == fs) | i_ft & (e2n == ft)));
assign fwdfa = e3w & (e3n == fs); // forward fpu e3d to fp a
assign fwdfb = e3w & (e3n == ft); // forward fpu e3d to fp b
assign wfpr  = i_lwc1 & wpcir; // fp regfile write enable
assign fwdla = mwfpr & (mrn == fs); // forward mmo to fp a
```

```
    assign fwdlb = mwfpr & (mrn == ft); // forward mmo to fp b
    assign stall_lwc1 = ewfpr & (i_fs & (ern == fs) | i_ft & (ern == ft));
    assign swfp  = i_swc1; // select signal
    assign fwdf  = swfp & e3w & (ft == e3n); // forward to id  stage
    assign fwdfe = swfp & e2w & (ft == e2n); // forward to exe stage
    assign stall_swc1 = swfp & e1w & (ft == e1n); // stall
    assign wpcir = ~(stall_div_sqrt | stall_others);
    wire stall_others = stall_lw | stall_fp | stall_lwc1 | stall_swc1 | st;
    assign fc = fop & {3{~stall_others}};
    assign wf = i_fs & wpcir;
    assign fasmds = i_fs;
endmodule
```

10.5 存储器模块及CPU/FPU的测试

本节给出存储器模块的代码、CPU的测试程序和仿真波形。

10.5.1 指令存储器和数据存储器

指令存储器模块:

```
module inst_mem (a,inst);
    input [31:0] a;
    output [31:0] inst;
    lpm_rom  lpm_rom_component  (.address(a[7:2]),.q(inst));
    defparam lpm_rom_component.lpm_width     = 32,
             lpm_rom_component.lpm_widthad   =  6,
             lpm_rom_component.lpm_numwords  = "unused",
             lpm_rom_component.lpm_file      = "inst_mem.mif",
             lpm_rom_component.lpm_indata    = "unused",
             lpm_rom_component.lpm_outdata   = "unregistered",
             lpm_rom_component.lpm_address_control = "unregistered";
endmodule
```

数据存储器模块:

```
module data_mem (we,addr,datain,clk,inclk,outclk,dataout);
    input  [31:0] addr,datain;
    input         clk,we,inclk,outclk;
    output [31:0] dataout;
    wire          write_enable = we & ~clk;
    lpm_ram_dq ram (.data(datain),.address(addr[6:2]),
                    .we(write_enable),.inclock(inclk),
                    .outclock(outclk),.q(dataout));
    defparam   ram.lpm_width    = 32;
    defparam   ram.lpm_widthad =  5;
    defparam   ram.lpm_indata   = "registered";
```

```
    defparam    ram.lpm_outdata = "registered";
    defparam    ram.lpm_file     = "data_mem.mif";
    defparam    ram.lpm_address_control = "registered";
endmodule
```

10.5.2 CPU/FPU 的测试程序

以下是 mif 格式的测试程序代码。主要目的是测试与 FPU 有关的指令的执行，但为了保证 CPU 的正确性，IU 指令的测试代码也列在后半部分，基本上与第 8 章的代码相同。浮点部分的代码用来测试与 lwc1 指令的数据相关、浮点加减乘法指令之间的数据相关、swc1 指令与浮点运算指令的数据相关、浮点除法和开方指令以及它们之间的数据相关等。

```
DEPTH = 64;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
[0..3F] : 00000000; % Range--Every address from 0 to 3F = 00000000       %
 0 : 00000820; %(00)        add    r1,  r0, r0  # address of data[0]    %
 1 : C4200000; %(04)        lwc1   f0,   0(r1)  # load fp data          %
 2 : C4210050; %(08)        lwc1   f1,  80(r1)  # load fp data          %
 3 : C4220054; %(0C)        lwc1   f2,  84(r1)  # load fp data          %
 4 : C4230058; %(10)        lwc1   f3,  88(r1)  # load fp data          %
 5 : C424005C; %(14)        lwc1   f4,  92(r1)  # load fp data          %
 6 : 46002100; %(18)        add.s  f4,  f4, f0  # f4: stall 1           %
 7 : 460418C1; %(1C)        sub.s  f3,  f3, f4  # f4: stall 2           %
 8 : 46022082; %(20)        mul.s  f2,  f4, f2  # mul                   %
 9 : 46040842; %(24)        mul.s  f1,  f1, f4  # mul                   %
 A : E4210070; %(28)        swc1   f1, 112(r1)  # f1: stall 1           %
 B : E4220074; %(2C)        swc1   f2, 116(r1)  # store fp data         %
 C : E4230078; %(30)        swc1   f3, 120(r1)  # store fp data         %
 D : E424007C; %(34)        swc1   f4, 124(r1)  # store fp data         %
 E : 20020004; %(38)        addi   r2,  r0,  4  # counter               %
 F : C4230000; %(3C) l3:    lwc1   f3,   0(r1)  # load fp data          %
10 : C4210050; %(40)        lwc1   f1,  80(r1)  # load fp data          %
11 : 46030840; %(44)        add.s  f1,  f1, f3  # stall 1               %
12 : 46030841; %(48)        sub.s  f1,  f1, f3  # stall 2               %
13 : E4210030; %(4C)        swc1   f1,  48(r1)  # stall 1               %
14 : C4050004; %(50)        lwc1   f5,  04(r0)  # load fp data          %
15 : C4060008; %(54)        lwc1   f6,  08(r0)  # load fp data          %
16 : C408000C; %(58)        lwc1   f8,  12(r0)  # load fp data          %
17 : 460629C3; %(5C)        div.s  f7,  f5, f6  # div                   %
18 : 46004244; %(60)        sqrt.s f9,  f8      # sqrt                  %
19 : 46004A84; %(64)        sqrt.s f10, f9      # sqrt                  %
```

```
1A : 2042FFFF; %(68)          addi   r2,  r2, -1  # counter - 1            %
1B : 1440FFF3; %(6C)          bne    r2,  r0, 13  # finish?                %
1C : 20210004; %(70)          addi   r1,  r1,  4  # address+4, DELAY SLOT %
1D : 3c010000; %(74) iu_test:lui  r1, 0           # address of data[0]     %
1E : 34240050; %(78)          ori  r4, r1, 80     # address of data[0]     %
1F : 0c000038; %(7C) call:    jal  sum            # call function          %
20 : 20050004; %(80) dslot1: addi r5, r0,  4      # DELYED SLOT(DS)        %
21 : ac820000; %(84) return: sw   r2, 0(r4)       # store result           %
22 : 8c890000; %(88)          lw   r9, 0(r4)      # check sw               %
23 : 01244022; %(8C)          sub  r8, r9, r4     # sub: r8 <-- r9 - r4    %
24 : 20050003; %(90)          addi r5, r0,  3     # counter                %
25 : 20a5ffff; %(94) loop2:  addi r5, r5, -1      # counter - 1            %
26 : 34a8ffff; %(98)          ori  r8, r5, 0xffff # zero-extend: 0000ffff %
27 : 39085555; %(9C)          xori r8, r8, 0x5555 # zero-extend: 0000aaaa %
28 : 2009ffff; %(A0)          addi r9, r0, -1     # sign-extend: ffffffff %
29 : 312affff; %(A4)          andi r10, r9, 0xffff# zero-extend: 0000ffff %
2A : 01493025; %(A8)          or   r6, r10, r9    # or:  ffffffff          %
2B : 01494026; %(AC)          xor  r8, r10, r9    # xor: ffff0000          %
2C : 01463824; %(B0)          and  r7, r10, r6    # and: 0000ffff          %
2D : 10a00003; %(B4)          beq  r5, r0, shift  # if r5 = 0, goto shift %
2E : 00000000; %(B8) dslot2: nop                  # DS                     %
2F : 08000025; %(BC)          j    loop2          # jump loop2             %
30 : 00000000; %(C0) dslot3: nop                  # DS                     %
31 : 2005ffff; %(C4) shift:  addi r5, r0, -1      # r5   = ffffffff        %
32 : 000543c0; %(C8)          sll  r8, r5, 15     # <<  15 = ffff8000      %
33 : 00084400; %(CC)          sll  r8, r8, 16     # <<  16 = 80000000      %
34 : 00084403; %(D0)          sra  r8, r8, 16     # >>> 16 = ffff8000      %
35 : 000843c2; %(D4)          srl  r8, r8, 15     # >>  15 = 0001ffff      %
36 : 08000036; %(D8) finish: j    finish         # dead loop              %
37 : 00000000; %(DC) dslot4: nop                  # DS                     %
38 : 00004020; %(E0) sum:    add  r8, r0, r0     # sum                    %
39 : 8c890000; %(E4) loop:   lw   r9, 0(r4)      # load data              %
3A : 01094020; %(E8)          add  r8, r8, r9     # sum                    %
3B : 20a5ffff; %(EC)          addi r5, r5, -1     # counter - 1            %
3C : 14a0fffc; %(F0)          bne  r5, r0, loop   # finish?                %
3D : 20840004; %(F4) dslot5: addi r4, r4,  4      # address + 4, DS        %
3E : 03e00008; %(F8)          jr   r31            # return                 %
3F : 00081000; %(FC) dslot6: sll  r2, r8, 0       # move res. to v0, DS    %
END ;
```

数据存储器的内容：

```
DEPTH = 32;           % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
```

```
BEGIN
[0..1F] : 00000000; % Range--Every address from 0 to 1F = 00000000 %
   0 :    BF800000; % (00) 1 01111111 00..0 fp -1 %
   1 :    40800000; % (04) %
   2 :    40000000; % (08) %
   3 :    41100000; % (0C) %
  14 :    40C00000; % (50) 0 10000001 10..0 data[0] 4.5 %
  15 :    41C00000; % (54) 0 10000011 10..0 data[1] %
  16 :    43C00000; % (58) 0 10000111 10..0 data[2] %
  17 :    47C00000; % (5C) 0 10001111 10..0 data[3] %
END ;
```

10.5.3 CPU / FPU 的仿真波形

仿真波形很长, 本书不可能全部列出, 图 10.23 ~ 图 10.29 示出重点部分的波形。

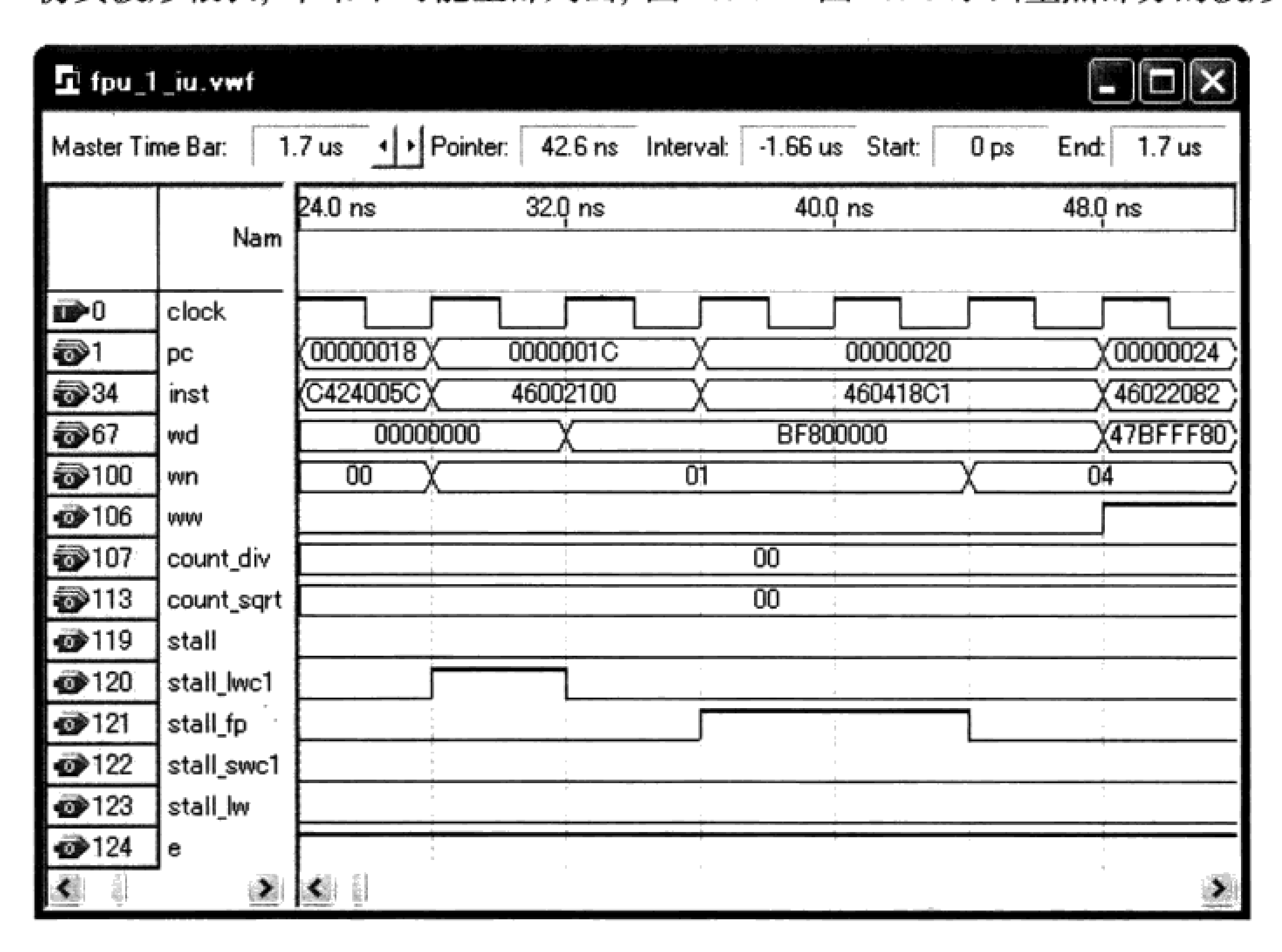


图 10.23 CPU 仿真波形图 (lwc1 和 add.s)

图 10.23 中在 24ns ~ 28ns (PC = 18) 期间取 add.s f4, f4, f0 指令 (IF)。它与 PC = 14 处的指令 lwc1 f4, 92(r1) 数据相关，因此在 28ns ~ 32ns (ID) 期间 stall_lwc1 = 1，流水线暂停一个周期。PC = 1C 处的指令 sub.s f3, f3, f4 又与 add.s 数据相关，暂停两个周期 (stall_fp = 1)。add.s 的浮点加法结果 (47C00000 + BF800000 = 47BFFF80) 在 48ns ~ 50ns 写回 (WB)。图 10.24 中，PC = 28 处的指令 swc1 f1, 112(r1) 与 mul.s 数据相关，在 56ns ~ 60ns 期间 stall_swc1 = 1，暂停一个周期。

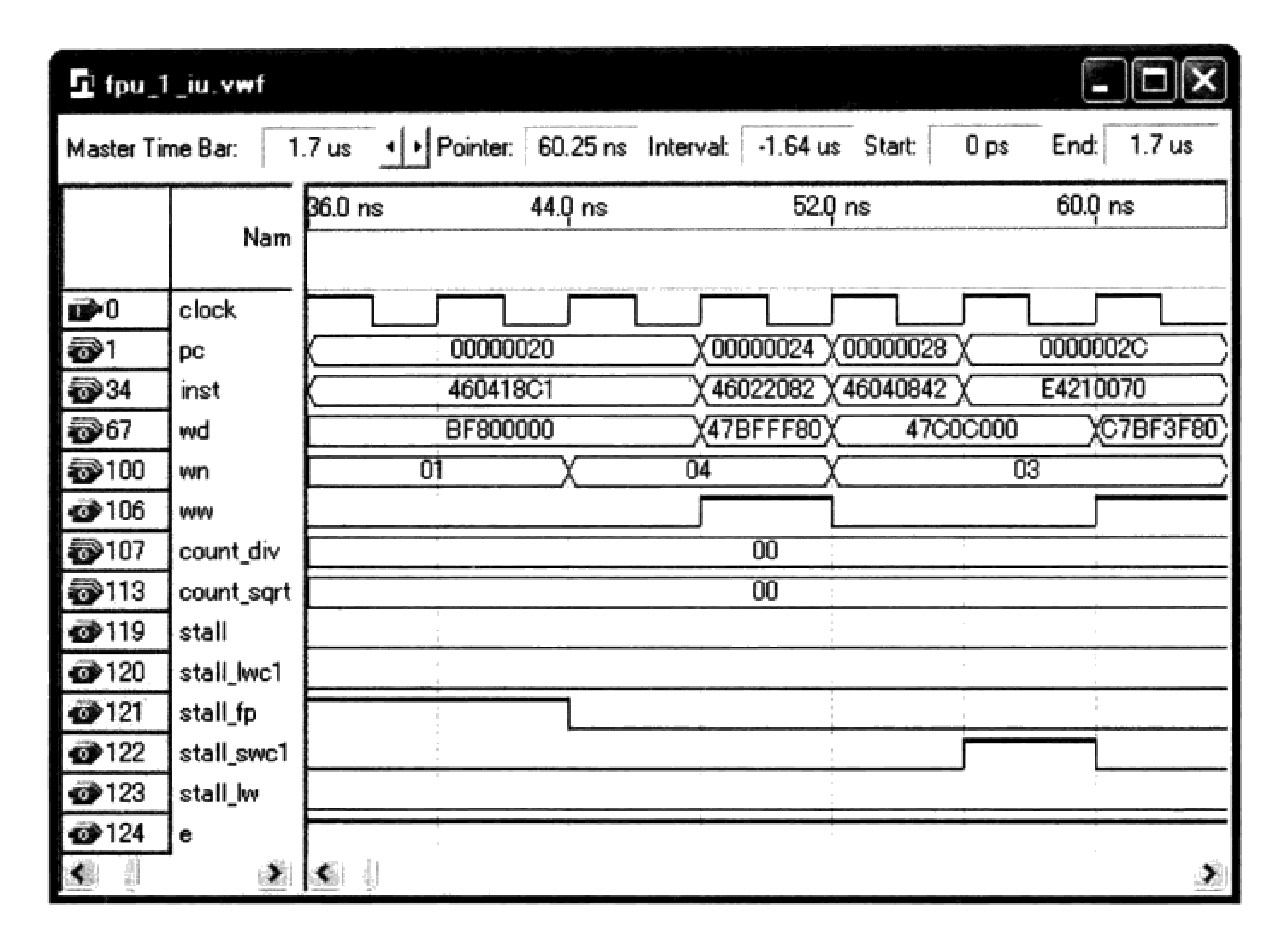


图 10.24　CPU 仿真波形图 (mul.s 和 swc1)

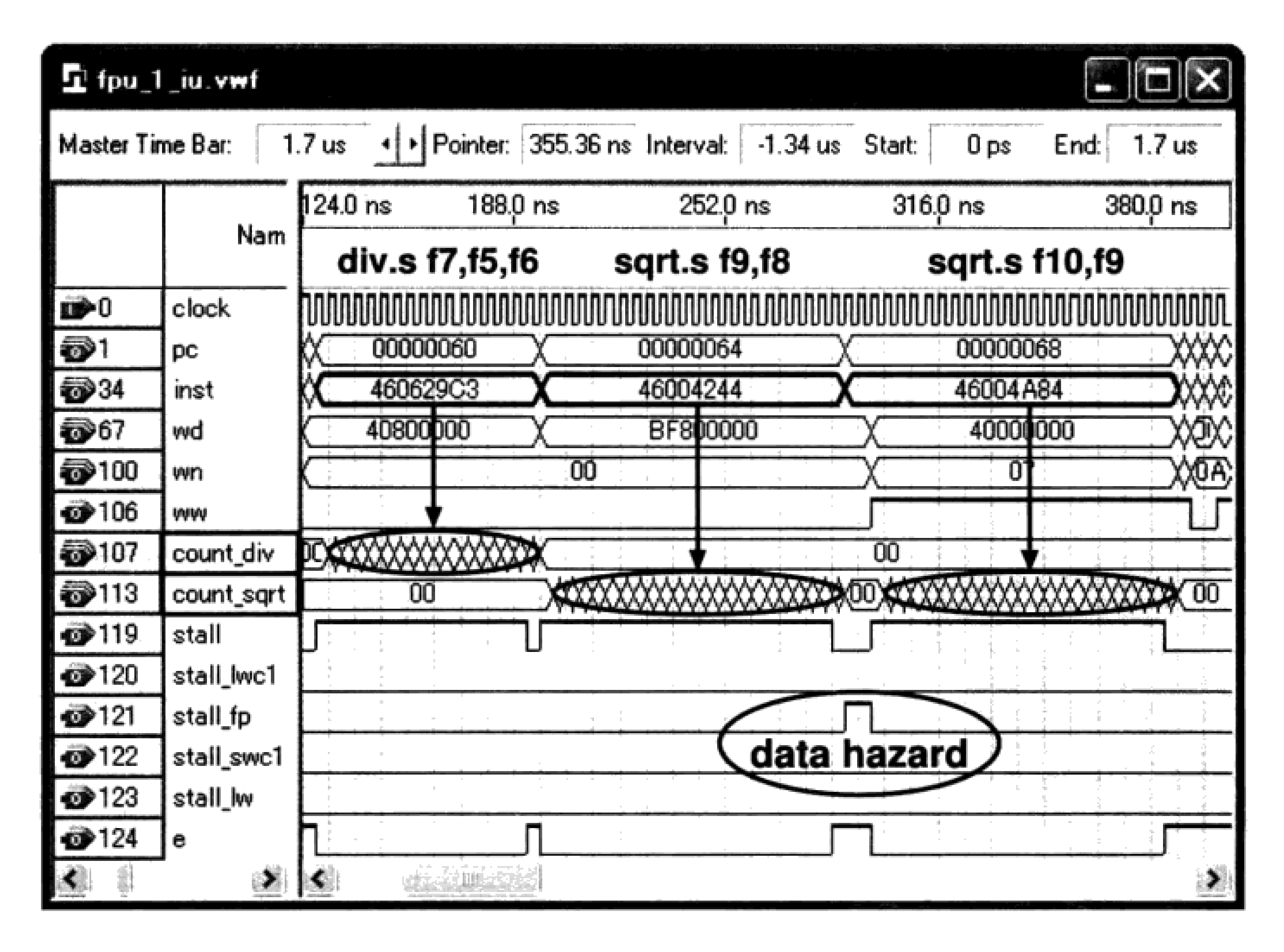


图 10.25　CPU 仿真波形图 (div.s、sqrt.s、sqrt.s)

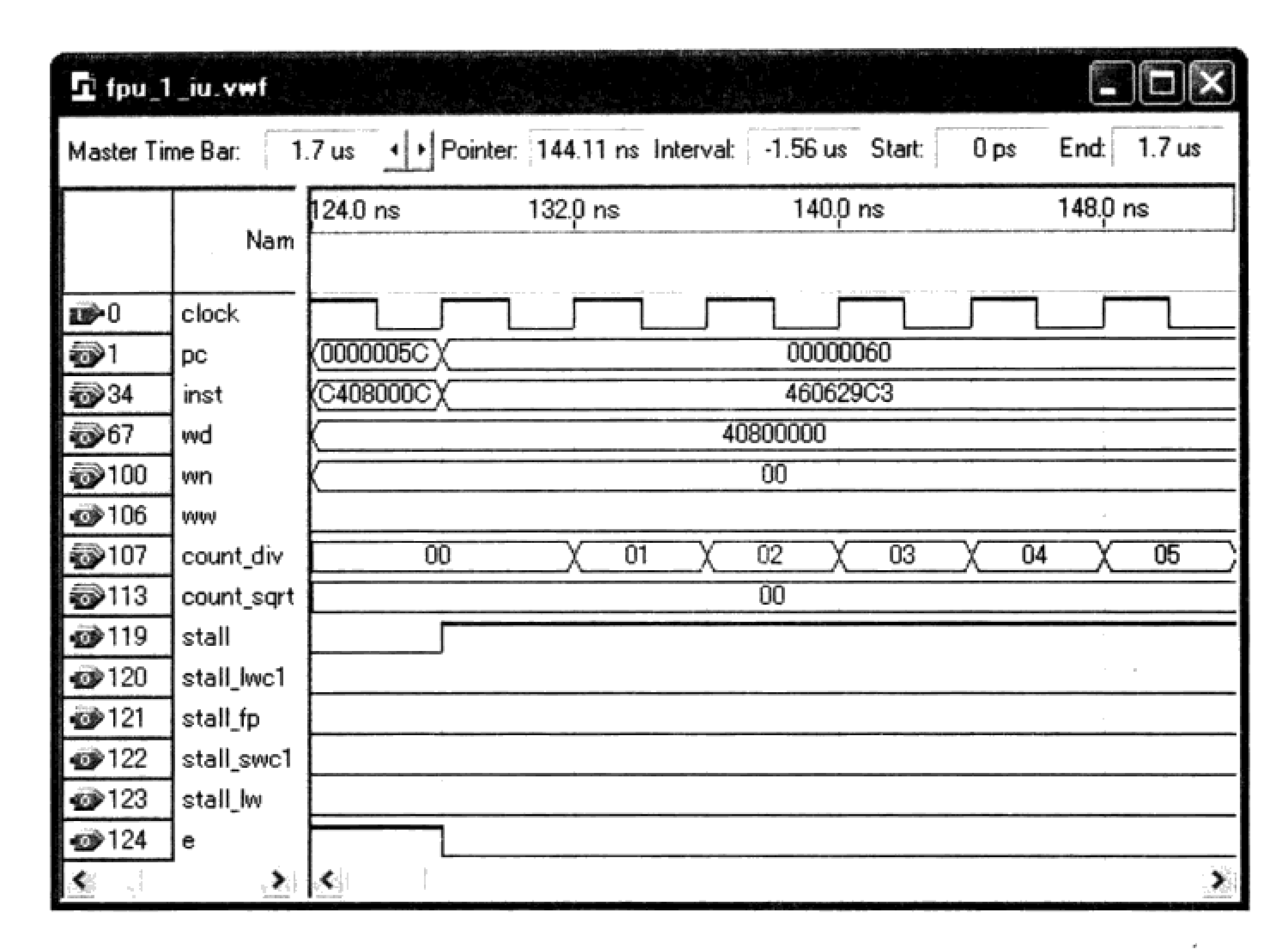


图 10.26　CPU 仿真波形图 (div.s 开始)

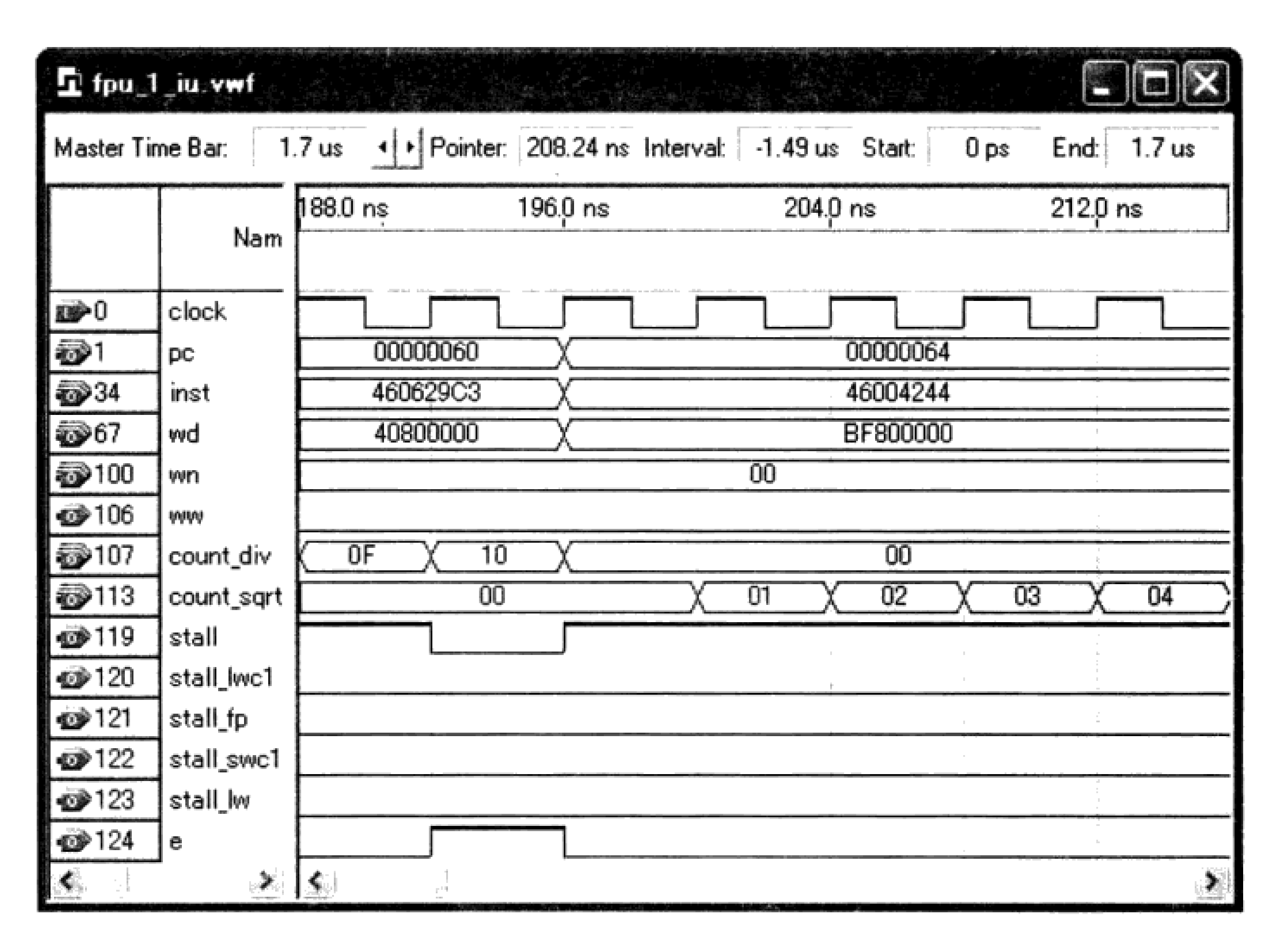


图 10.27　CPU 仿真波形图 (div.s 结束)

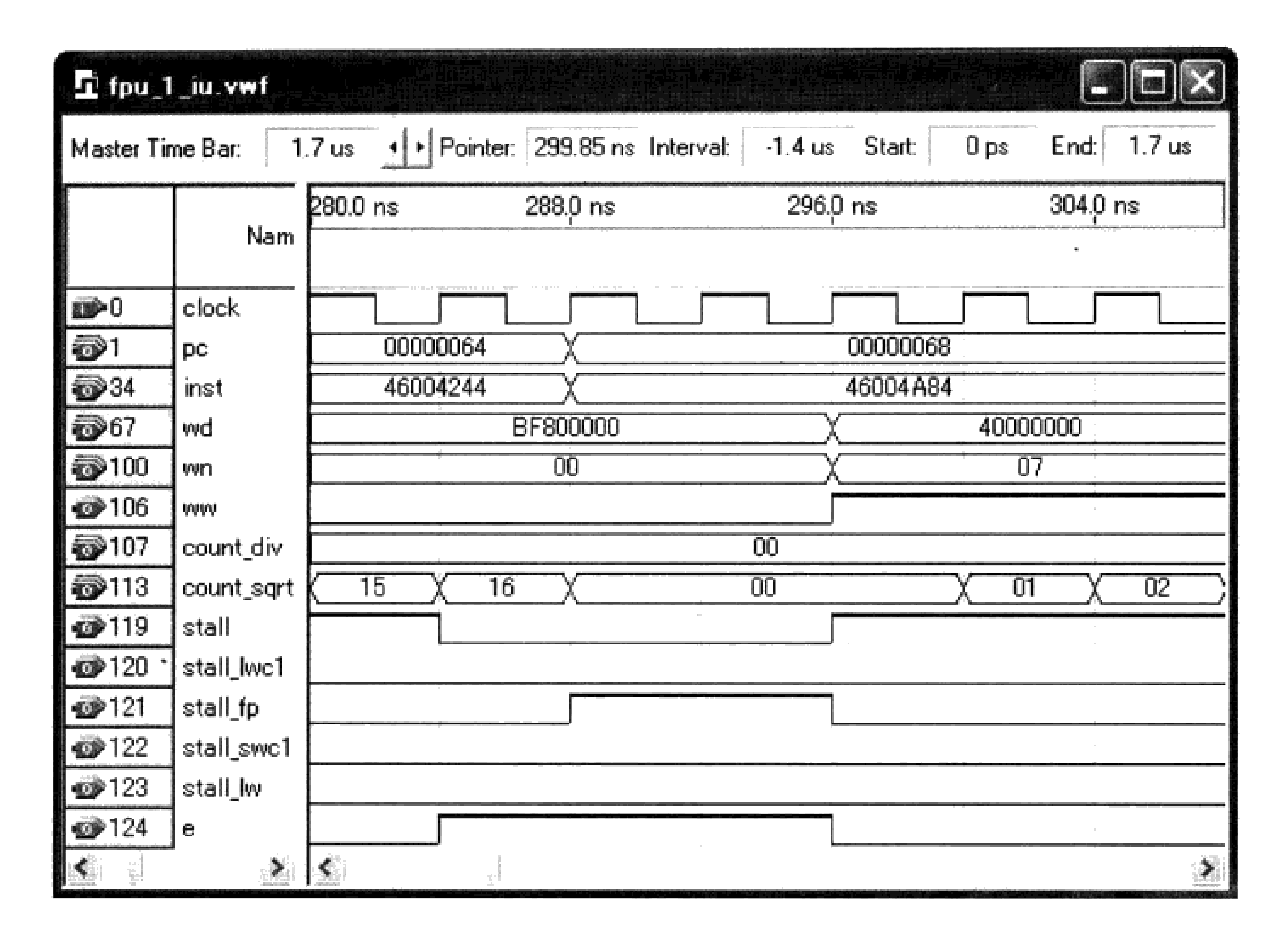


图 10.28　带有 FPU 的流水线 CPU 仿真波形图 (sqrt.s 开始)

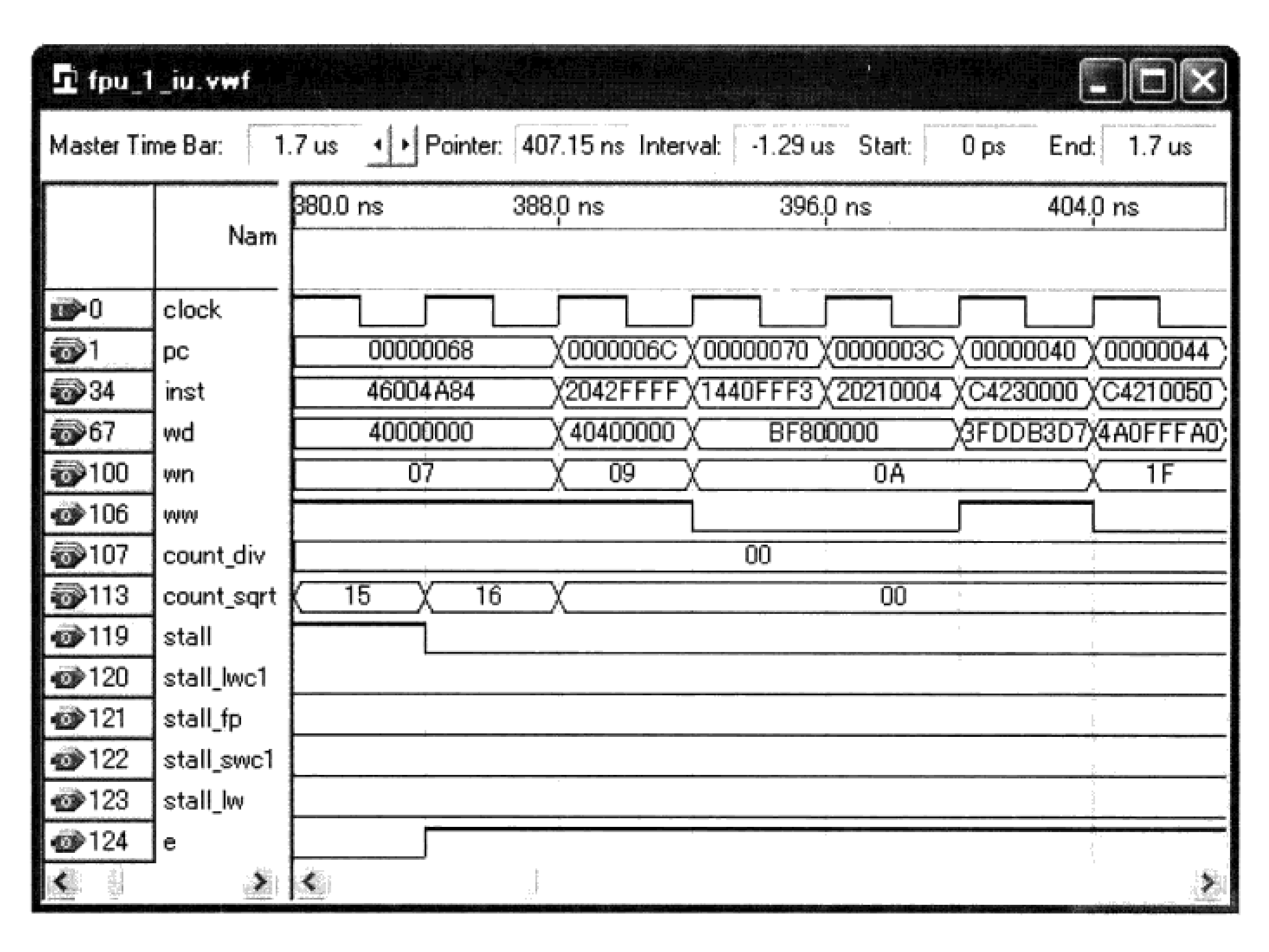


图 10.29　带有 FPU 的流水线 CPU 仿真波形图 (sqrt.s 结束)

图 10.25 示出 div.s、sqrt.s、sqrt.s 指令波形的总体图。由于两条开方指令数据相关，因此 stall_fp = 1。图 10.26 中 PC = 5C 处的指令是 div.s f7, f5, f6。它启动除法计数器，stall = 1，一直持续到计数器的值等于 0F (见图 10.27)。紧接着的是开方指令 sqrt.s f9, f8，它也启动计数器，stall = 1，一直持续到计数器的值等于十六进制的 15 (见图 10.28)。然后又是一条开方指令：sqrt.s f10, f9，并且与上一条开方指令数据相关 (stall_fp = 1)。启动计数器，stall = 1，一直持续到 384ns 处 (见图 10.29)。然后把一条除法指令和两条开方指令的执行结果分别写入浮点寄存器 07、09 和 0A 中 (ww = 1)。

10.6 习题

1. 使用本章 CPU 能够执行的指令，试着编写一些有实际意义的、使用浮点运算的程序，将其转换成 mif 格式，进行仿真并检验结果。
2. 试着把浮点寄存器与整数寄存器之间的数据传输指令 mfc1 和 mtc1 以及浮点数与整数之间的转换指令 cvt.s.w 和 cvt.w.s 加入 CPU 的设计中。

```
mfc1     rt, fs  # rt <-- fs
mtc1     rt, fs  # fs <-- rt
cvt.s.w  fd, fs  # fd <-- convert_and_round(fs)
cvt.w.s  fd, fs  # fd <-- convert_and_round(fs)
```

   指令格式如下所示。

指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	意 义
mfc1	010001	00000	rt	fs	00000	000000	取浮点寄存器字
mtc1	010001	00100	rt	fs	00000	000000	存浮点寄存器字
cvt.s.w	010001	10100	00000	fs	fd	100000	整数转成浮点数
cvt.w.s	010001	10000	00000	fs	fd	100000	浮点数转成整数

3. 试考虑加入异常 (包括浮点计算结果异常) 和中断处理。
4. 实现浮点除法和开方指令以完全的流水线方式执行：每个周期可以接收一条浮点除法或开方指令，不停流水线，除非数据相关。试使用以下三种方法：
   1) 使用不恢复余数算法实现流水线的单精度浮点除法和开方会降低电路的复杂性且易于实现。
   2) 使用较大的 ROM 得到足够精度的 x_0，然后只用一次 Newton 迭代。
   3) 增加一条新的指令，它只实现一次 Newton 迭代。如果一次单精度浮点除法或开方运算需要 3 次迭代，则连续执行 3 次这条指令 (双精度浮点除法或开方运算执行 4 次)。

5. 参考图 10.30，设计一个简单的超标量 (Superscalar) 流水线 CPU，实现浮点指令与整数指令的并行执行。图中的 W 代表流水线的结果写回级，占用半个时钟周期。浮点部件的执行级占用三个周期 (E1 ～ E3)，除法和开方迭代在 ID 级完成 (停流水线)。

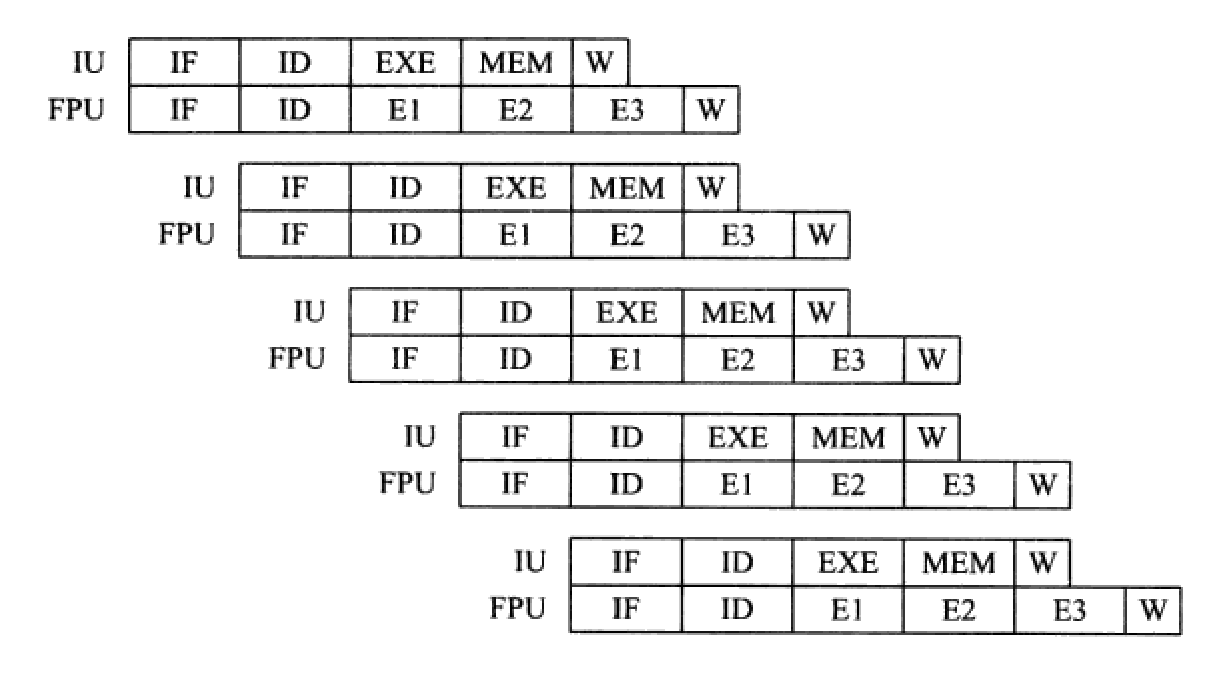


图 10.30 简单的超标量 CPU 流水线时序图

6. 设计一个复杂的超标量流水线 CPU，使浮点加、减、乘、除和开方指令能并行执行。整数部件也可有多个执行部件，使得不相关的整数指令也能并行执行。

第 11 章　多线程 CPU 及其 Verilog HDL 设计

多核 (Multi-Core) 及多线程 (Multithreading) 技术已被世界知名计算机公司 (比如 Intel、IBM、AMD、Fujitsu 等) 广泛用于 CPU 的设计中。本章主要讨论多线程 CPU 的原理及设计。

11.1　多线程 CPU 概述

本节简要介绍多线程的基本概念以及能够并行执行多线程的 CPU 的基本结构。

11.1.1　多线程 CPU 的基本概念

一个线程是指一段程序在执行过程中不依赖其他线程产生的数据。见图 11.1，线程 D 和线程 E 从理论上讲可以并行执行，但线程 F 只有等线程 D 和线程 E 均执行完才能开始执行。

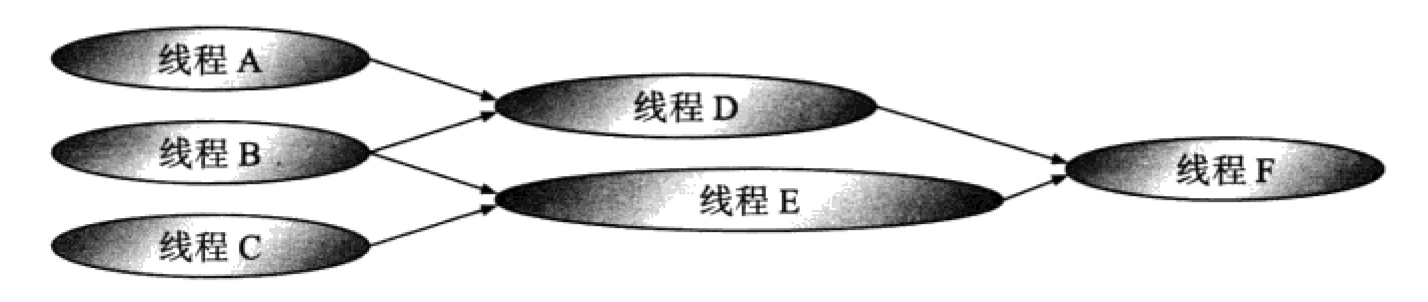


图 11.1　线程的基本概念

虽然有些线程可以并行执行，但如果 CPU 不提供相应的硬件支持，多个本来可以并行执行的线程也只能轮流被执行，如图 11.2(a) 所示。传统的 CPU 就属于这种情况。从任意一个时间点来看，CPU 实际上只在执行一个线程。我们称其为单线程 CPU。依线程切换的方式不同，单线程 CPU 又可分为两种，这里不再详细描述。

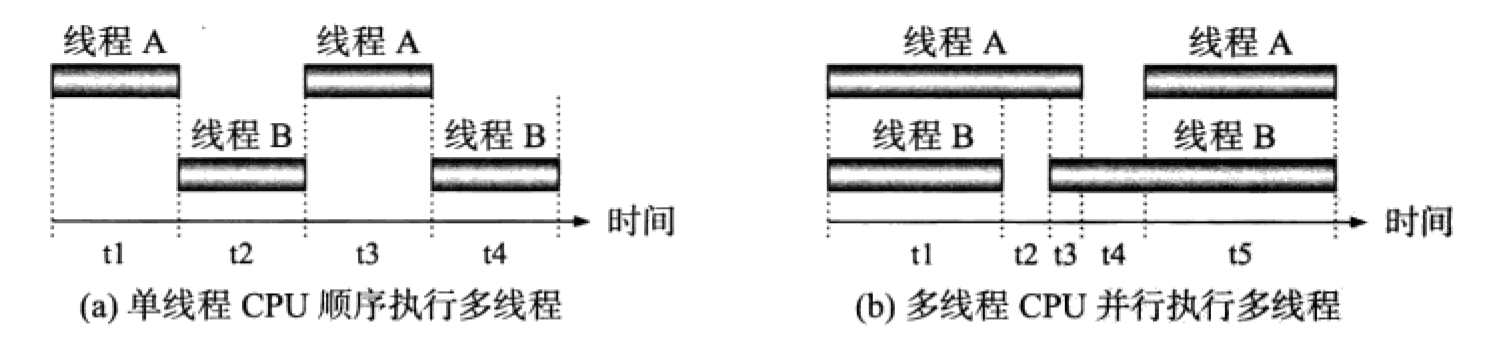


图 11.2　多线程 CPU 的基本概念

而多线程 CPU 则不同。如图 11.2(b) 所示，在时间段 t1、t3 和 t5 中，两个线程可以并行地被执行。这样的 CPU 需要有多个程序计数器以及能够完成运算功能的多个功能部件 (Functional Units)。与单纯的多核技术相比，多线程 CPU 的好处是功能部件可以得到充分的利用 (多个线程共享功能部件) [6, 7, 34, 2]。

11.1.2 多线程 CPU 的基本结构

图 11.3 示出的是最简单的双线程 CPU 的一个例子。图中有两个程序计数器 (PC0 和 PC1) 及两个寄存器堆 (Register File 0 和 Register File 1)，它能同时执行两个线程，相当于有两个逻辑意义上的 CPU。三个功能部件 (两个 ALU 和一个 FPU) 被两个逻辑 CPU 所共享。当两个线程同时要求使用 FPU时，其中一个的要求得到满足，而另一个必须等待。下面一节详细讨论线程的选择方法。

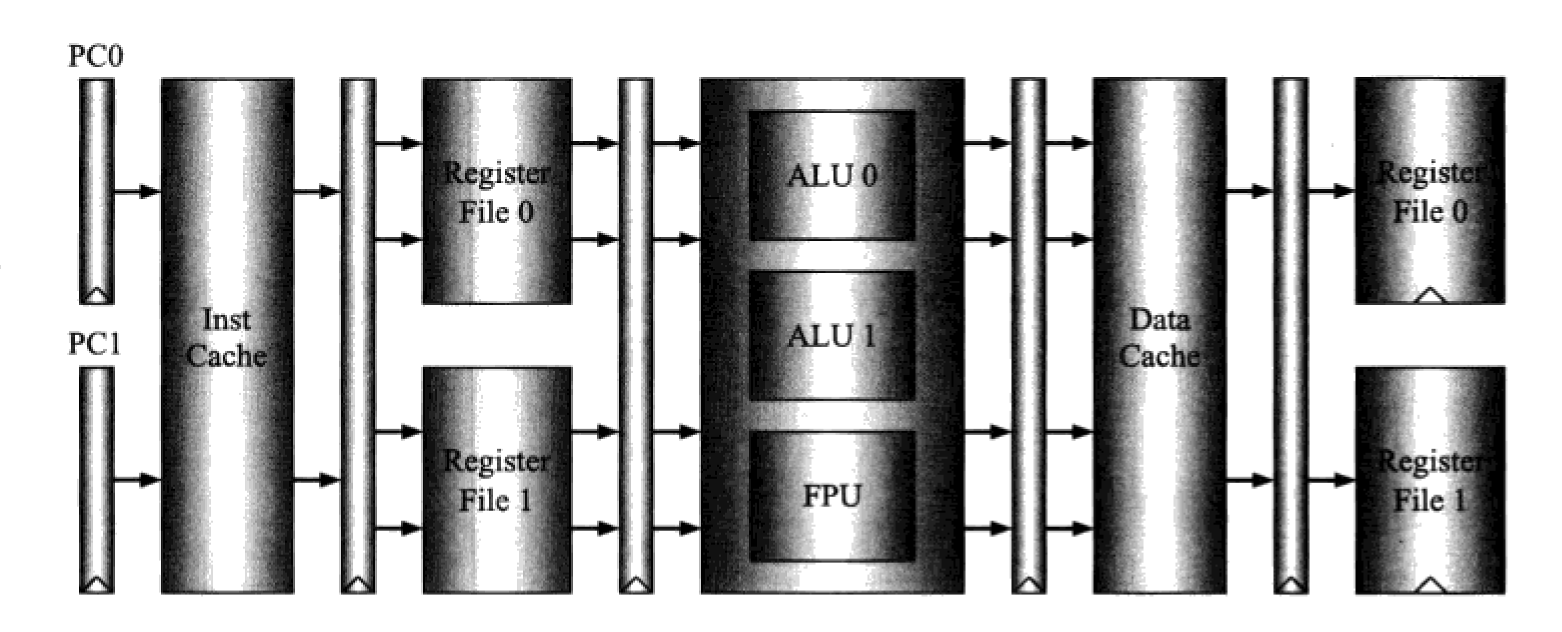


图 11.3 多线程 CPU 的基本结构

11.2 多线程 CPU 设计

本节给出多线程 CPU 的线程选择方法、详细电路及具体的 Verilog HDL 代码。

11.2.1 线程的选择方法

假设多线程 CPU 中某种功能部件只有一个，问题就出现了：当多个线程同时要使用这个功能部件时，CPU 到底选择哪个线程？我们可以用一个简单的计数器来选择一个线程。如果想让某些线程有较高的优先级得到执行，我们可以增大计数器的计数范围，多选一些计数值让那些线程有优先被执行的权力。比如一个 3 位的计数器可以使两个线程的优先级之比为 5 : 3。本章给出双线程 CPU 的设计方法，而且令两个线程有相同的优先级。

图 11.4 示出的是双线程 CPU 选择线程的电路，模块名为 selthread。信号 fasmds0 和 fasmds1 分别表示线程 0 和线程 1 是否有使用浮点部件的请求；信号 dt 表示在 ID 级被选中的线程；信号 st0 和 st1 分别表示是否要停线程 0 和线程 1 的流水线。左边的一个 dffe 和一个非门构成一个计数器，输出为 cnt。右边 4 个 dffe 是流水线寄存器，它们的输出分别对应流水线的 E1、E2、E3 和 WB 级。中间的 thread_sel 模块是电路的主要部分，它的真值表在表 11.1 中给出。

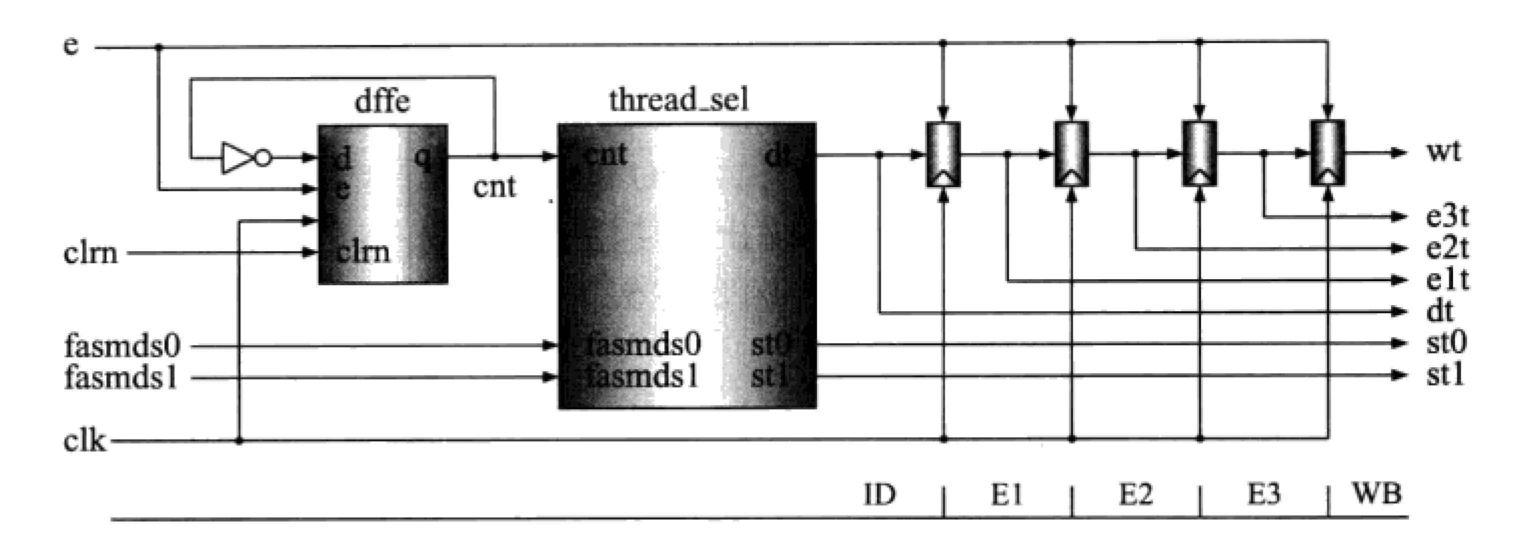


图 11.4 线程的选择电路 selthread

表 11.1 线程的选择方法 (图 11.4 中的 thread_sel 模块)

输 入			输 出		
cnt	fasmds0	fasmds1	dt	st0	st1
0	1	0	0	0	0
0	1	1	0	0	1
0	0	1	1	0	0
1	0	1	1	0	0
1	1	1	1	1	0
1	1	0	0	0	0
x	0	0	0	0	0

表 11.1 的内容应该不难理解。当 fasmds0 和 fasmds1 均为 1 时 (都有请求)，根据 cnt 选择线程 dt 并封锁线程 $\overline{dt}$。st0 和 st1 分别是线程 0 和线程 1 的暂停信号。由此，我们得到 3 个输出信号的逻辑表达式：

```
dt  = ~fasmds0 & fasmds1 | cnt & fasmds1;
st0 =  cnt & fasmds0 & fasmds1;
st1 = ~cnt & fasmds0 & fasmds1;
```

11.2.2 多线程 CPU 的详细电路

我们已经在第 10 章介绍了带有 FPU 的流水线 CPU 的设计，以此为基础，本小节详细介绍双线程 CPU 的设计。

我们的双线程 CPU 有两个 ALU 和一个 FPU。本来两个 ALU 应该被两个线程所共享，为了简化设计，我们的双线程 CPU 为每个线程分配一个固定的 ALU。这相当于 CPU 中有两个独立的 IU 和一个共享的 FPU。

如果两个线程中的任何一个线程完全没有浮点指令时，两个线程可以互不干扰地并行执行。只有两个线程同时要执行浮点指令时，才会出现竞争 FPU 的情况。因此，多线程 CPU 设计的重点和难点在于如何解决对共享部件的竞争的问题。

我们已经在 11.2.1 小节介绍了当竞争出现时如何选择线程的电路。从被选中的线程的浮点寄存器堆读出的数据要被送到 FPU，因此我们需要在 FPU 的数据输入端使用一个二选一多路器 (Multiplexer)。另外，FPU 的计算结果也要被写入相应线程的浮点寄存器堆中，因此我们要在 FPU 的输出端使用一个类似于反向多路器的电路，我们暂且称其为分配器 (Demultiplexer)，见图 11.5。

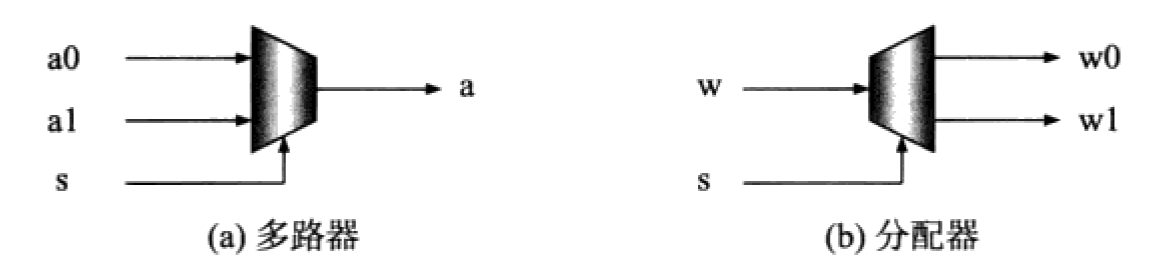


图 11.5 分配器与多路器的比较

比如浮点寄存器堆的写使能信号就要用到分配器。分配器输出信号的逻辑表达式如下，其中的信号 s 指明把输入信号 w 分配给谁：w0 还是 w1。

```
w0 = ~s & w;
w1 =  s & w;
```

有些信号并不需要分配，比如目的寄存器号和计算结果。有了多路器和分配器这两件神器，一个双线程的 CPU 的结构大致就出来了，见图 11.6。图中有两个整数部件 iu0 和 iu1；两个浮点寄存器堆以及相应的用于数据内部前推的多路器。图中 fpu 模块左边的是多路器 (mux)，右边的是分配器 (demux)。分配器的选择信号来自于 11.2.1 小节描述的 selthread 模块，即图中右下角部分。

要被分配的信号处在不同的流水线级：stall、e1w、e2w、e3w 和 ww 分别处在流水线的 ID、E1、E2、E3 和 WB 级，相应的分配信号分别为 dt、e1t、e2t、e3t 和 wt。分配器的输出 ww0 和 ww1 分别接到线程 0 和线程 1 的浮点寄存器堆的写使能端，其余的接到相应的 iu。selthread 模块的输出信号 st0 和 st1 也接到相应的 iu，用于暂停流水线，暂停的原因是 FPU 的资源竞争。

多路器 mux 的选择信号是 ID 级的 dt。多路器的两路输入分别来自两个线程，其中除了浮点数据来自浮点寄存器堆右面的多路器，其余的均来自整数部件。整数部件 iu0 和 iu1 是等价的，与我们在第 10 章描述的相同；fpu 模块也与第 10 章描述的相同。

11.2.3 多线程 CPU 的 Verilog HDL 代码

以下是双线程 CPU 的 Verilog HDL 代码，模块名为 fpu_2_iu，它实现图 11.6 的电路。该模块的输出信号较多，完全是为了测试方便而设置的。

```
module fpu_2_iu (resetn,memclock,clock,
                 pc0,inst0,ealu0,malu0,walu0,ww0,
```

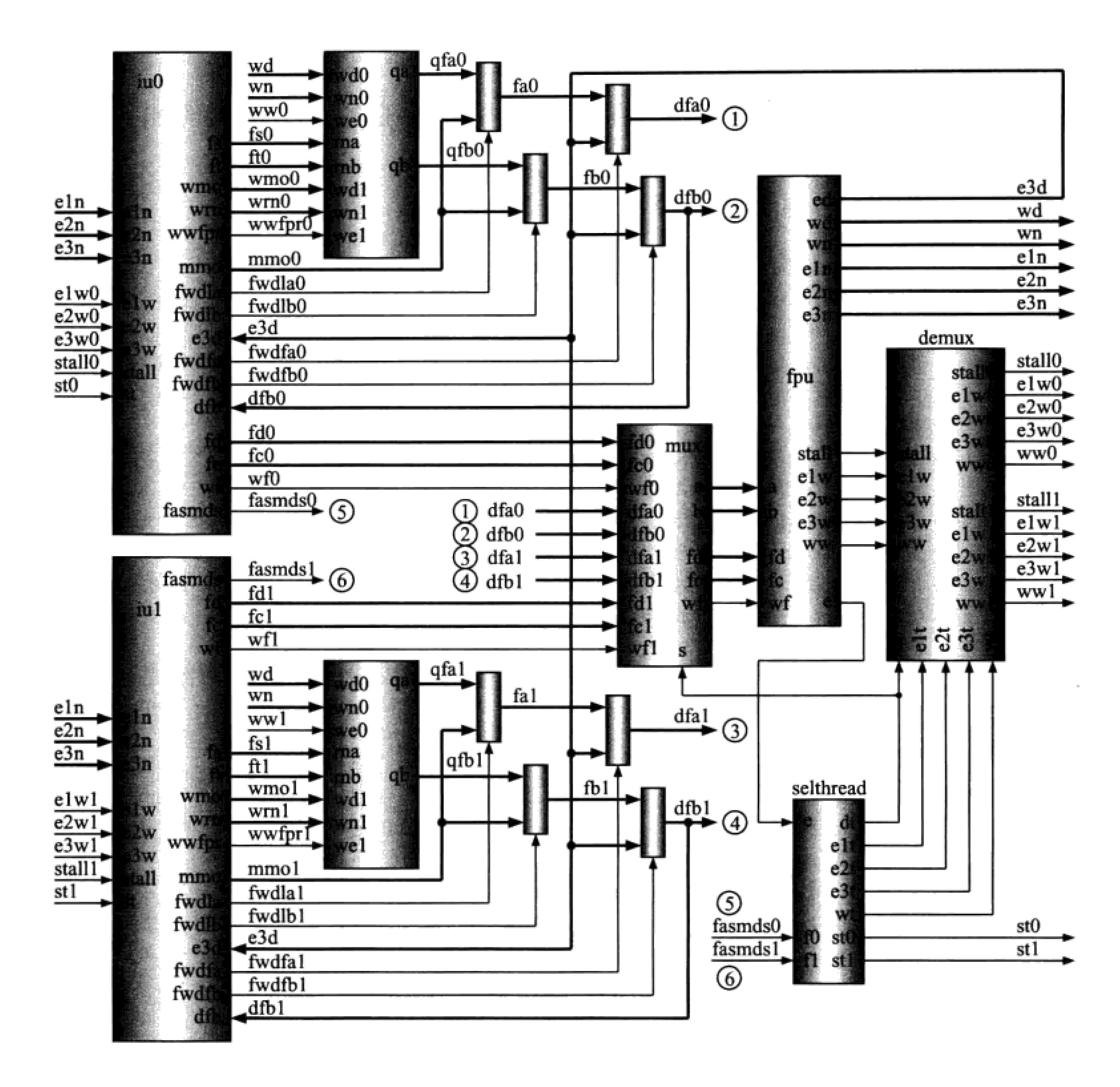


图 11.6 多线程 CPU 的模块图

```
              stall_lw0,stall_lwc10,stall_swc10,stall_fp0,stall0,st0,
              pc1,inst1,ealu1,malu1,walu1,ww1,
              stall_lw1,stall_lwc11,stall_swc11,stall_fp1,stall1,st1,
              wn,wd,count_div,count_sqrt,e1n,e2n,e3n,e3d,e);
input clock,memclock,resetn;
output [31:0] pc0,inst0,ealu0,malu0,walu0;
output [31:0] pc1,inst1,ealu1,malu1,walu1;
output ww0,stall_lw0,stall_lwc10,stall_swc10,stall_fp0,stall0,st0;
output ww1,stall_lw1,stall_lwc11,stall_swc11,stall_fp1,stall1,st1;
output e; // enable
output [31:0] e3d,wd;
output [4:0] e1n,e2n,e3n,wn;
```

```
output [4:0] count_div,count_sqrt;
```

以下是线程 0 (图 11.6 中的 iu0) 的代码，其中的 iu 模块已在第10 章给出：

```
wire [31:0] qfa0,qfb0,fa0,fb0,dfa0,dfb0,mmo0,wmo0;
wire [4:0]  fs0,ft0,fd0,wrn0;
wire [2:0]  fc0;
wire        fwdla0,fwdlb0,fwdfa0,fwdfb0,wf0,fasmds0;
iu iu0 (e1n,e2n,e3n, e1w0,e2w0,e3w0, stall0,st0,
        dfb0,e3d, clock,memclock,resetn,
        fs0,ft0,wmo0,wrn0,wwfpr0,mmo0,fwdla0,fwdlb0,fwdfa0,fwdfb0,
        fd0,fc0,wf0,fasmds0,pc0,inst0,ealu0,malu0,walu0,
        stall_lw0,stall_fp0,stall_lwc10,stall_swc10);
regfile2w fpr0 (fs0,ft0,wd,wn,ww0,wmo0,wrn0,wwfpr0,
               ~clock,resetn,qfa0,qfb0);
mux2x32 fwd_f_load_a0 (qfa0,mmo0,fwdla0,fa0); // forward lwc1 to fp a
mux2x32 fwd_f_load_b0 (qfb0,mmo0,fwdlb0,fb0); // forward lwc1 to fp b
mux2x32 fwd_f_res_a0  (fa0,e3d,fwdfa0,dfa0);  // forward fp res to fp a
mux2x32 fwd_f_res_b0  (fb0,e3d,fwdfb0,dfb0);  // forward fp res to fp b
```

以下是线程 1 (图 11.6 中的 iu1) 的代码，其中的 iu 模块已在第10 章给出：

```
wire [31:0] qfa1,qfb1,fa1,fb1,dfa1,dfb1,mmo1,wmo1;
wire [4:0]  fs1,ft1,fd1,wrn1;
wire [2:0]  fc1;
wire        fwdla1,fwdlb1,fwdfa1,fwdfb1,wf1,fasmds1;
iu iu1 (e1n,e2n,e3n, e1w1,e2w1,e3w1, stall1,st1,
        dfb1,e3d, clock,memclock,resetn,
        fs1,ft1,wmo1,wrn1,wwfpr1,mmo1,fwdla1,fwdlb1,fwdfa1,fwdfb1,
        fd1,fc1,wf1,fasmds1,pc1,inst1,ealu1,malu1,walu1,
        stall_lw1,stall_fp1,stall_lwc11,stall_swc11);
regfile2w fpr1 (fs1,ft1,wd,wn,ww1,wmo1,wrn1,wwfpr1,
               ~clock,resetn,qfa1,qfb1);
mux2x32 fwd_f_load_a1 (qfa1,mmo1,fwdla1,fa1); // forward lwc1 to fp a
mux2x32 fwd_f_load_b1 (qfb1,mmo1,fwdlb1,fb1); // forward lwc1 to fp b
mux2x32 fwd_f_res_a1  (fa1,e3d,fwdfa1,dfa1);  // forward fp res to fp a
mux2x32 fwd_f_res_b1  (fb1,e3d,fwdfb1,dfb1);  // forward fp res to fp b
```

以下是共享的浮点部件 (图 11.6 中的 fpu)，代码已在第10 章给出：

```
wire [1:0] e1c,e2c,e3c;
fpu fp_unit (dfa,dfb,fc,wf,fd,1'b1,clock,resetn,e3d,wd,wn,ww,
             stall,e1n,e1w,e2n,e2w,e3n,e3w,
             e1c,e2c,e3c,count_div,count_sqrt,e);
```

多路器，选择线程 0 还是线程 1 (图 11.6 中的 mux)：

```
wire [31:0]  dfa,dfb; // fp inputs a and b
wire [4:0]   fd;      // fp destination register number
```

```
    wire [2:0]    fc;       // fp operation code
    wire          wf;       // fp register file write enable
    assign dfa  = dt? dfa1 : dfa0;
    assign dfb  = dt? dfb1 : dfb0;
    assign fd   = dt? fd1  : fd0;
    assign wf   = dt? wf1  : wf0;
    assign fc   = dt? fc1  : fc0;
```

分配器 (图 11.6 中的 demux):

```
    // demux: for thread 0;        for thread 1
    wire stall0 = stall & ~dt;     wire stall1 = stall &  dt;  // ID stage
    wire   e1w0 =   e1w & ~e1t;    wire   e1w1 =   e1w &  e1t; // E1 stage
    wire   e2w0 =   e2w & ~e2t;    wire   e2w1 =   e2w &  e2t; // E2 stage
    wire   e3w0 =   e3w & ~e3t;    wire   e3w1 =   e3w &  e3t; // E3 stage
    wire    ww0 =    ww & ~wt;     wire    ww1 =    ww &  wt;  // WB stage
```

线程选择信号及流水线暂停信号的产生 (图 11.6 中的 selthread，细节见图 11.4):

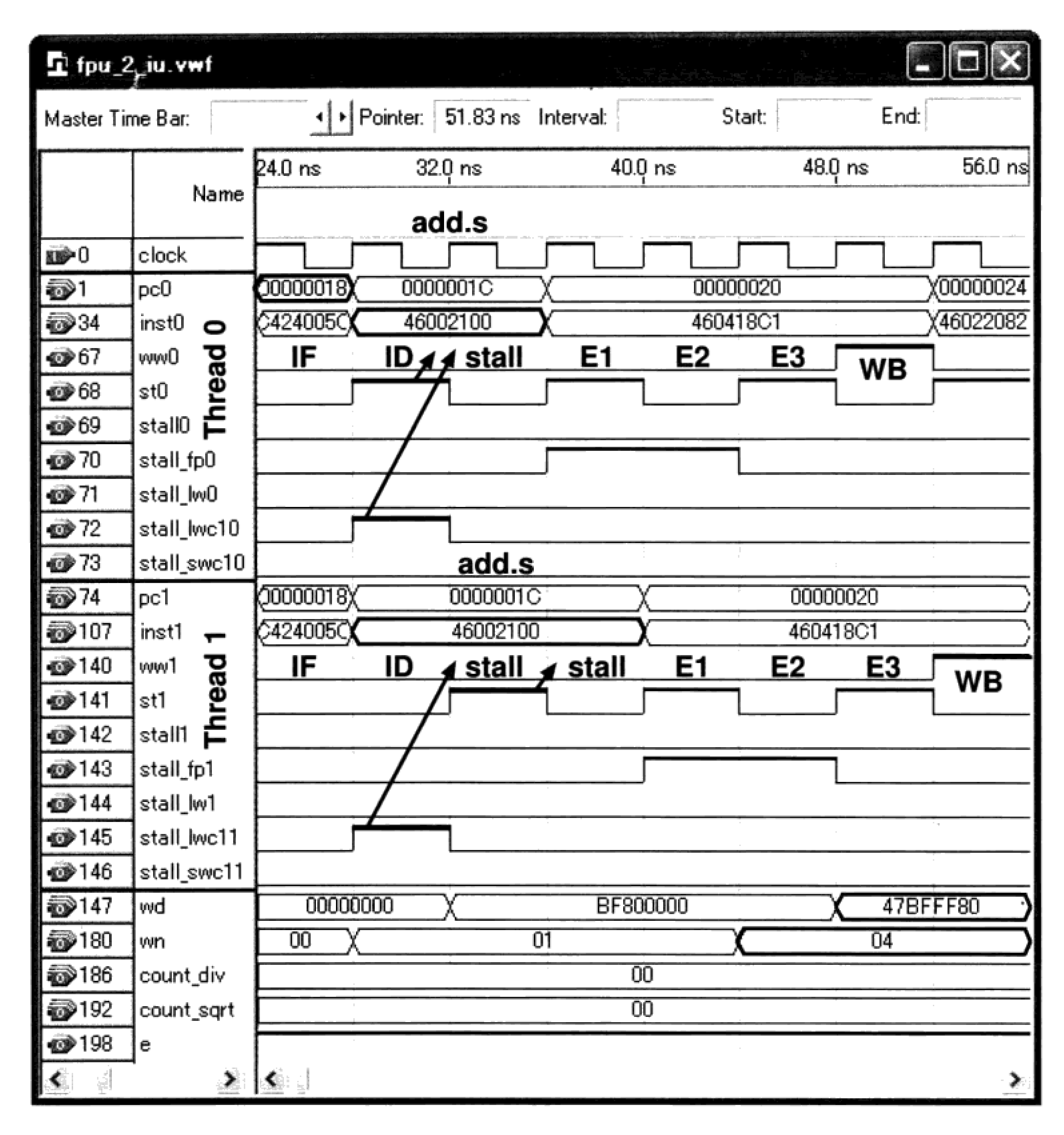
```
    // thread selection
    assign  st0 =  cnt & fasmds0 & fasmds1;              // stall thread 0
    assign  st1 = ~cnt & fasmds0 & fasmds1;              // stall thread 1
    wire    dt  = ~fasmds0 & fasmds1 | cnt & fasmds1; // selected thread

    // count for thread selection
    reg cnt;
    always @(negedge resetn or posedge clock)
      if (resetn == 0) begin
          cnt <= 0;
      end else if (e) begin // enable
          cnt <= ~cnt;
      end

    // pipelined thread info
    reg e1t,e2t,e3t,wt;
    always @(negedge resetn or posedge clock)
      if (resetn == 0) begin
          e1t <= 0;     e2t <= 0;     e3t <= 0;     wt  <= 0;
      end else if (e) begin // enable
          e1t <= dt;    e2t <= e1t;   e3t <= e2t;   wt  <= e3t;
      end
endmodule
```

11.3 多线程 CPU 的仿真波形

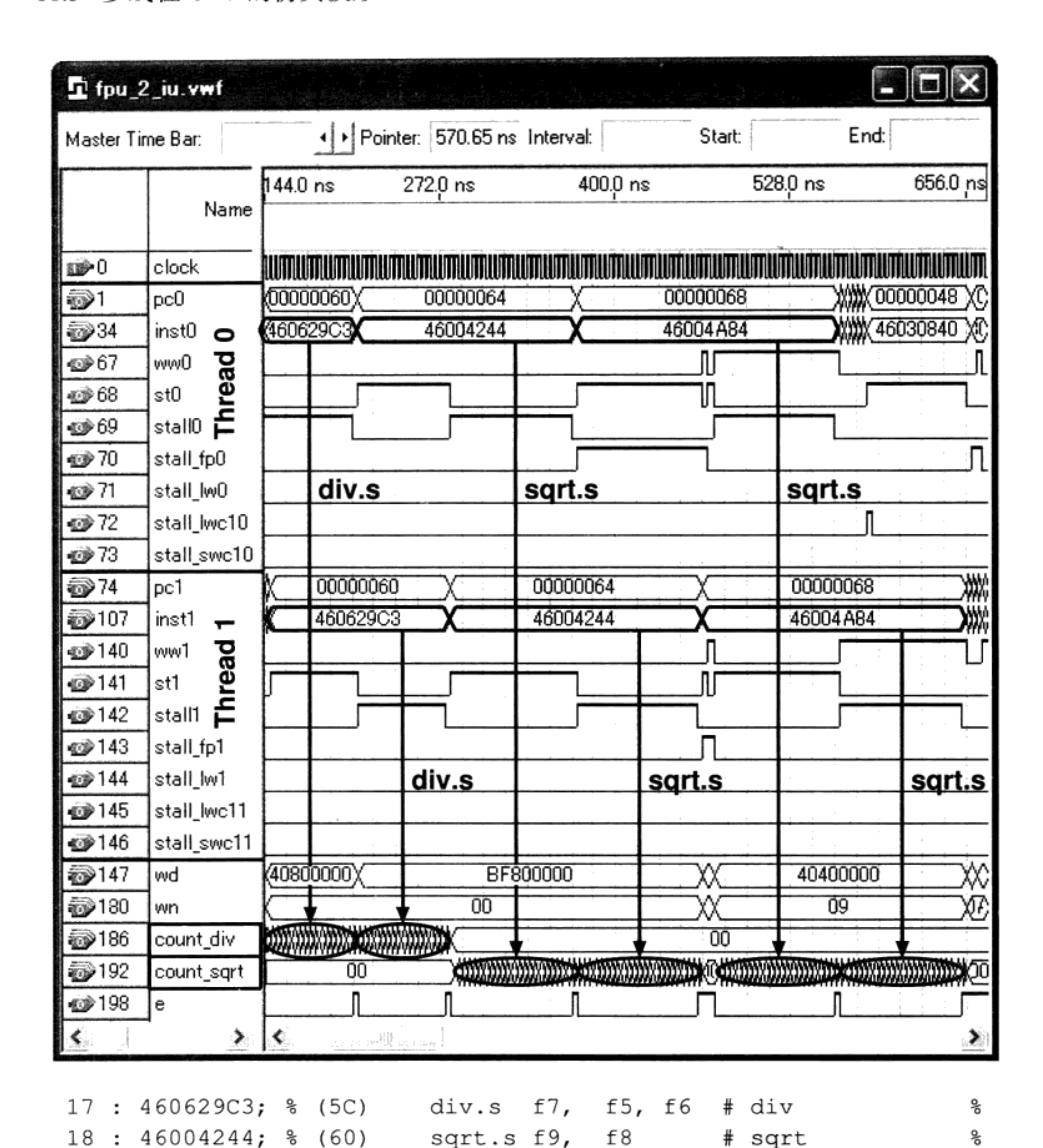
在本测试中，两个线程执行同样的程序。我们使用第 10 章的测试程序，重点给出浮点运算指令执行时的波形。以下图中最上部分是线程 0 的信号，中间部分是线程 1 的信号，最下部分是公用信号 (FPU 部分)。执行的指令列在波形的下面。




```
5 : C424005C; % (14)    lwc1   f4,   92(r1)   # load fp data    %
6 : 46002100; % (18)    add.s  f4,   f4, f0   # f4: stall 1     %
7 : 460418C1; % (1C)    sub.s  f3,   f3, f4   # f4: stall 2     %
```

图 11.7 多线程 CPU 仿真波形图 (线程 0 浮点加和线程 1 浮点加)

图 11.7 中线程 0 流水线的第一个暂停是由 add.s 指令与 lwc1 数据相关 (stall_lwc10) 或浮点部件资源冲突 (st0) 引起的。线程 1 的流水线暂停两个周期，一个是由 add.s 指令与 lwc1 数据相关引起的、另一个是由浮点部件资源冲突引起的。由于 sub.s 与 add.s 数据相关，流水线还要暂停两个周期。线程 0 和线程 1 分别在48ns ～ 50ns (前半个周期) 和 52ns ～ 54ns 处把 wd = 47BFFF80 写入各自的 wn = 04 浮点寄存器。




```
17 : 460629C3; % (5C)    div.s  f7,  f5, f6  # div     %
18 : 46004244; % (60)    sqrt.s f9,  f8      # sqrt    %
19 : 46004A84; % (64)    sqrt.s f10, f9      # sqrt    %
```

图 11.8 多线程 CPU 仿真波形图 (div.s、sqrt.s、sqrt.s 指令序列总体图)

图 11.8 示出的是执行 div.s、sqrt.s、sqrt.s 指令序列时的总体波形。下边密密麻麻的是除法计数器和开方计数器的值，在此期间完成 Newton-Raphson 迭代。两个线程轮流执行，先从线程 0 开始。线程 0 迭代时，stall0 为 1 (暂停自己的流水线等待迭代完成)、st1 为 1 (由于资源冲突而暂停对方的流水线)；线程 1 迭代时，stall1 为 1、st0 为 1。我们将陆续给出能够看清计数器值的详细的波形。

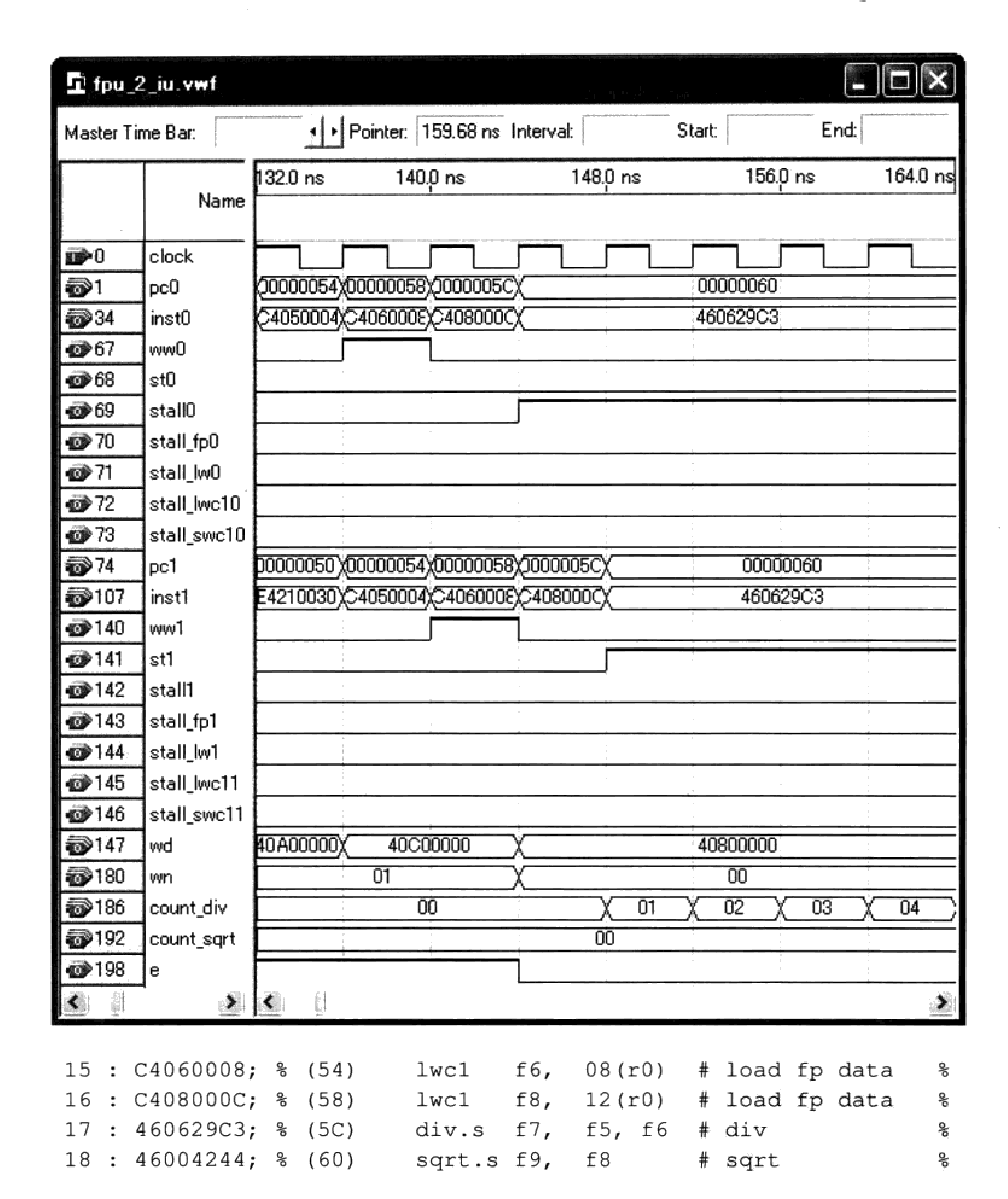


图 11.9 多线程 CPU 仿真波形图 (线程 0 浮点除开始)

图 11.9 示出的是线程 0 开始执行 div.s 指令时的波形。从 140ns (pc0 = 0000005C) 开始取 div.s f7, f5, f6 指令 (IF)；从 144ns 开始译码并进行除法的 Newton-Raphson 迭代 (ID)。在译码期间，stall0 = 1、除法计数器开始从 0 计数。在 148ns 处，线程 1 也对 div.s 指令译码，但这时线程 0 正在使用浮点部件，因此 st1 = 1。

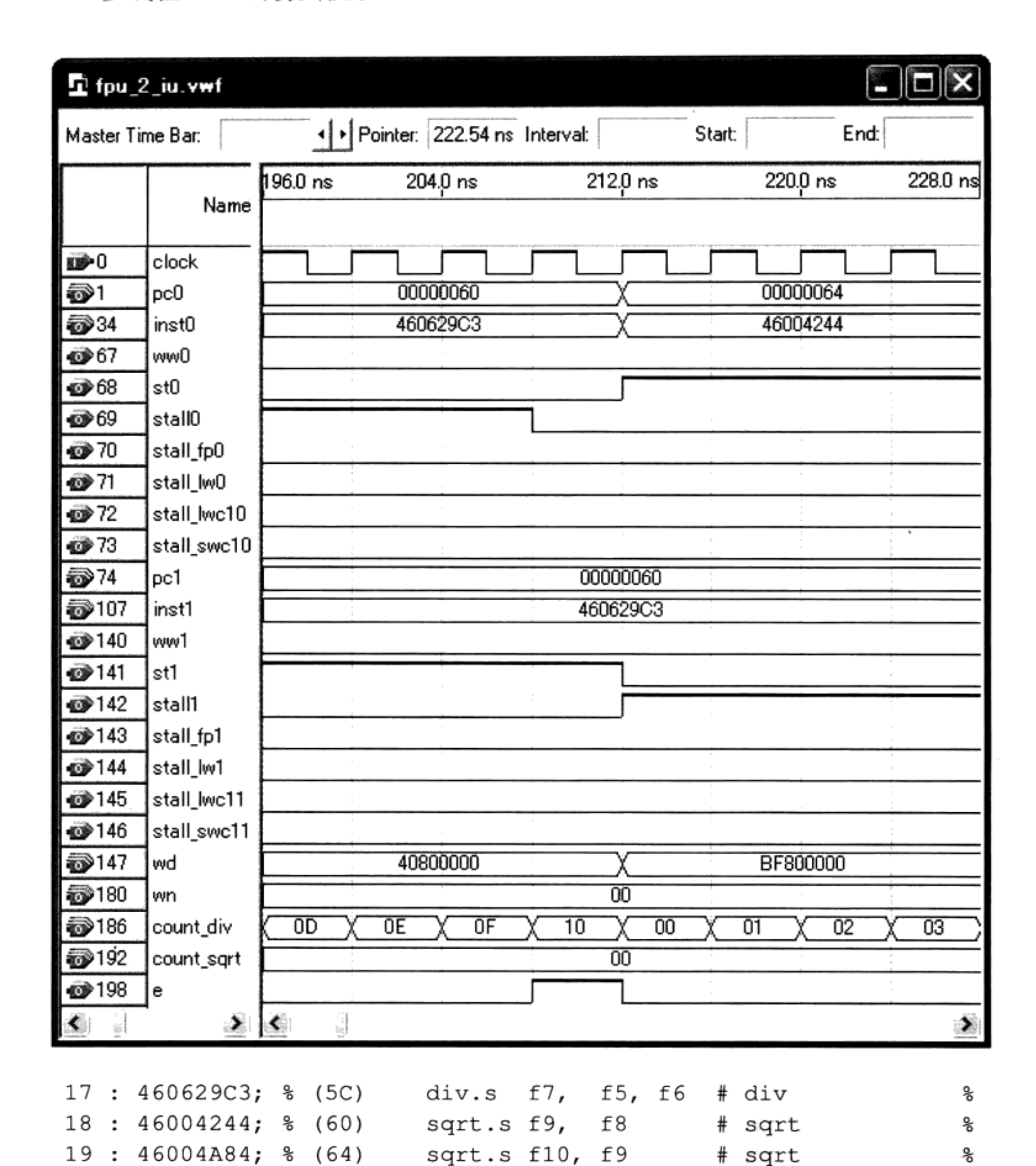
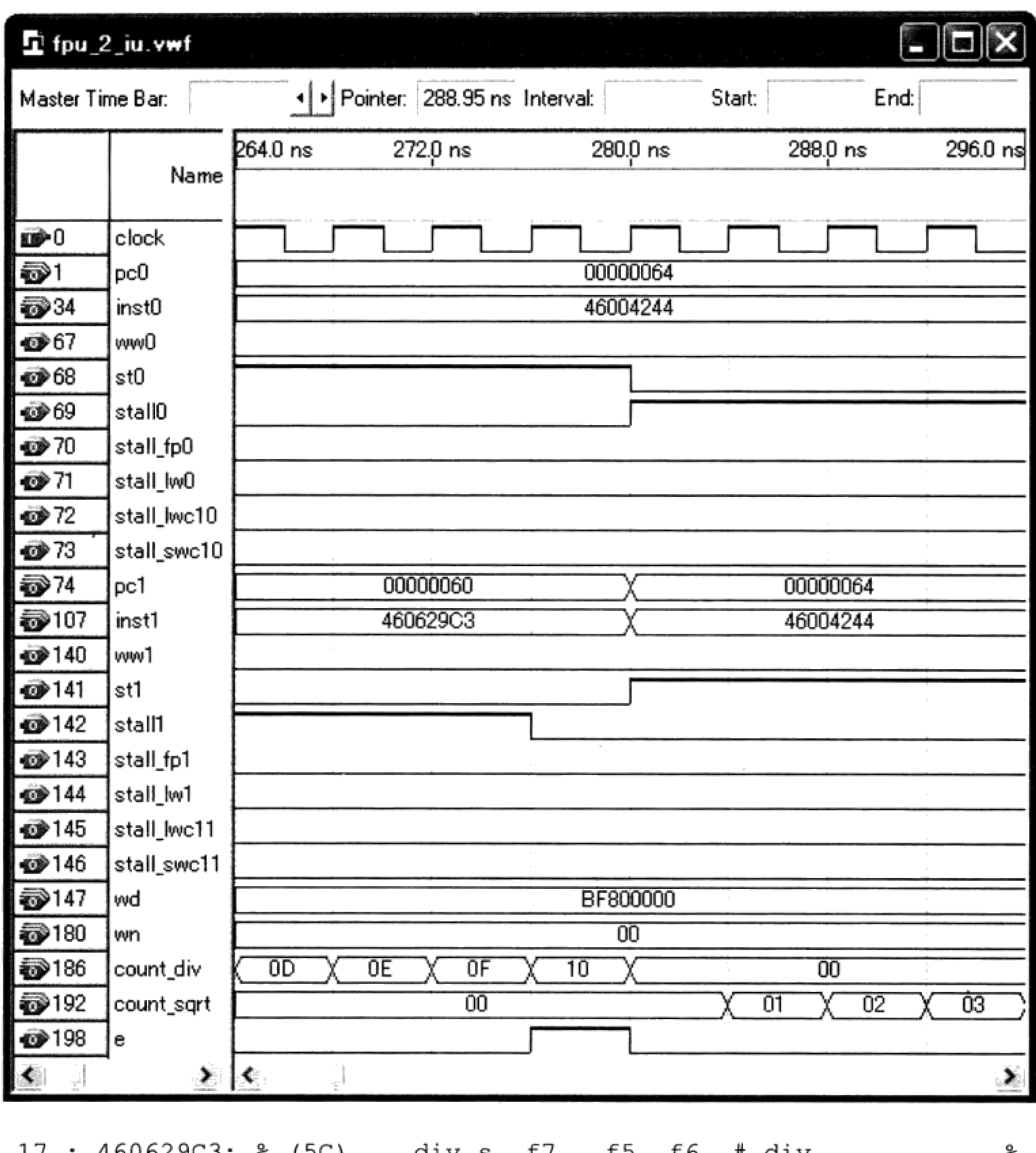


图 11.10 多线程 CPU 仿真波形图 (线程 1 浮点除开始)

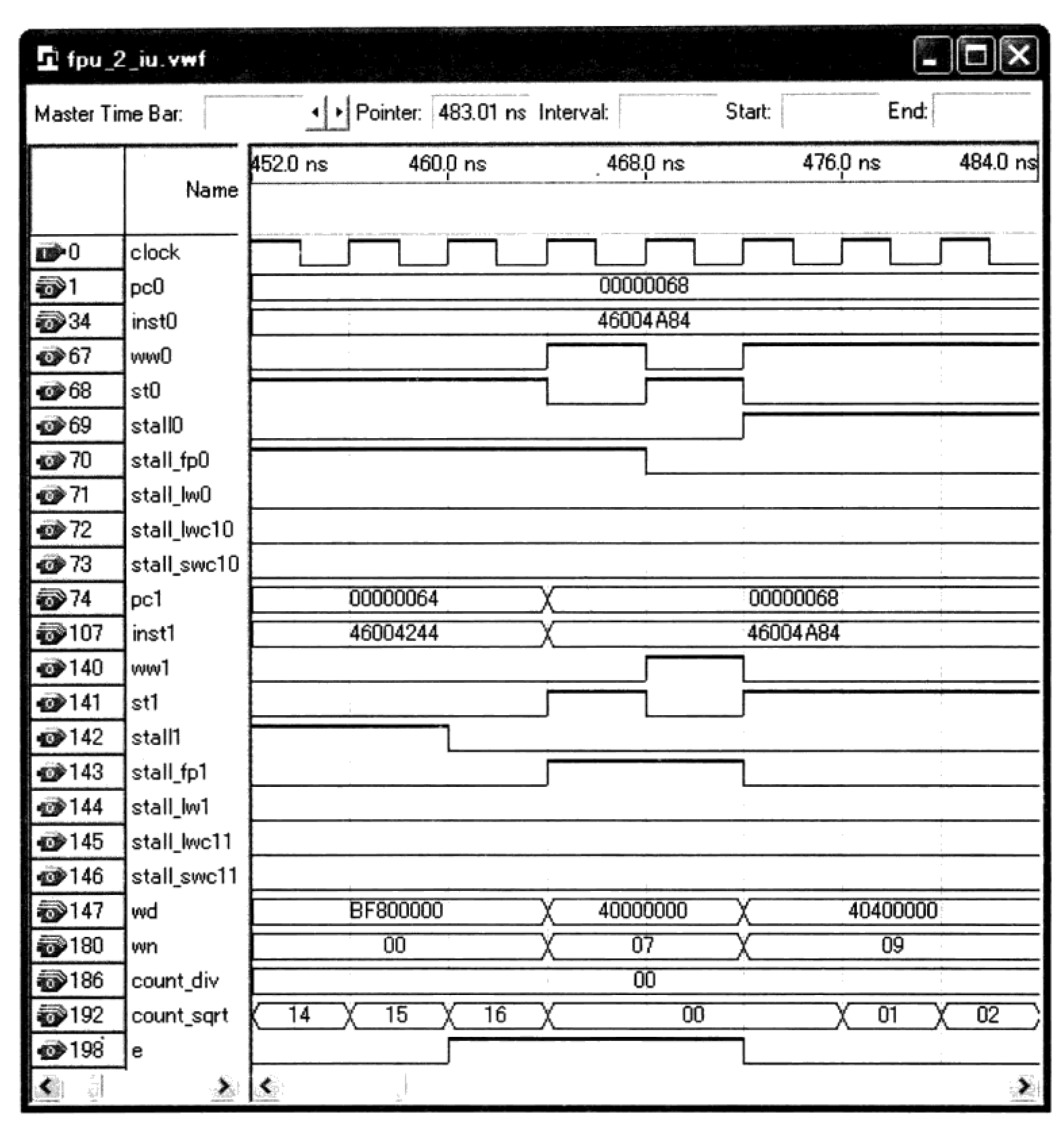
图 11.10 示出的是浮点部件连续执行需要迭代的指令 (除法指令或开方指令) 时的情况。图中的 208ns 处，stall0 变 0，导致使能信号 e 变 1，进而结束线程 0 的 div.s 指令的 ID 级。从 212ns 处开始，线程 1 进入 div.s 指令的 ID 级，stall1 = 1。与此同时，线程 0 试图对 sqrt.s 指令进行译码，但这时线程 1 正处在 div.s 指令的译码期间，线程 0 必须等待 (st0 = 1)。




```
17 : 460629C3; % (5C)    div.s  f7,  f5, f6  # div     %
18 : 46004244; % (60)    sqrt.s f9,  f8      # sqrt    %
19 : 46004A84; % (64)    sqrt.s f10, f9      # sqrt    %
```

图 11.11　多线程 CPU 仿真波形图 (线程 0 浮点开方开始)

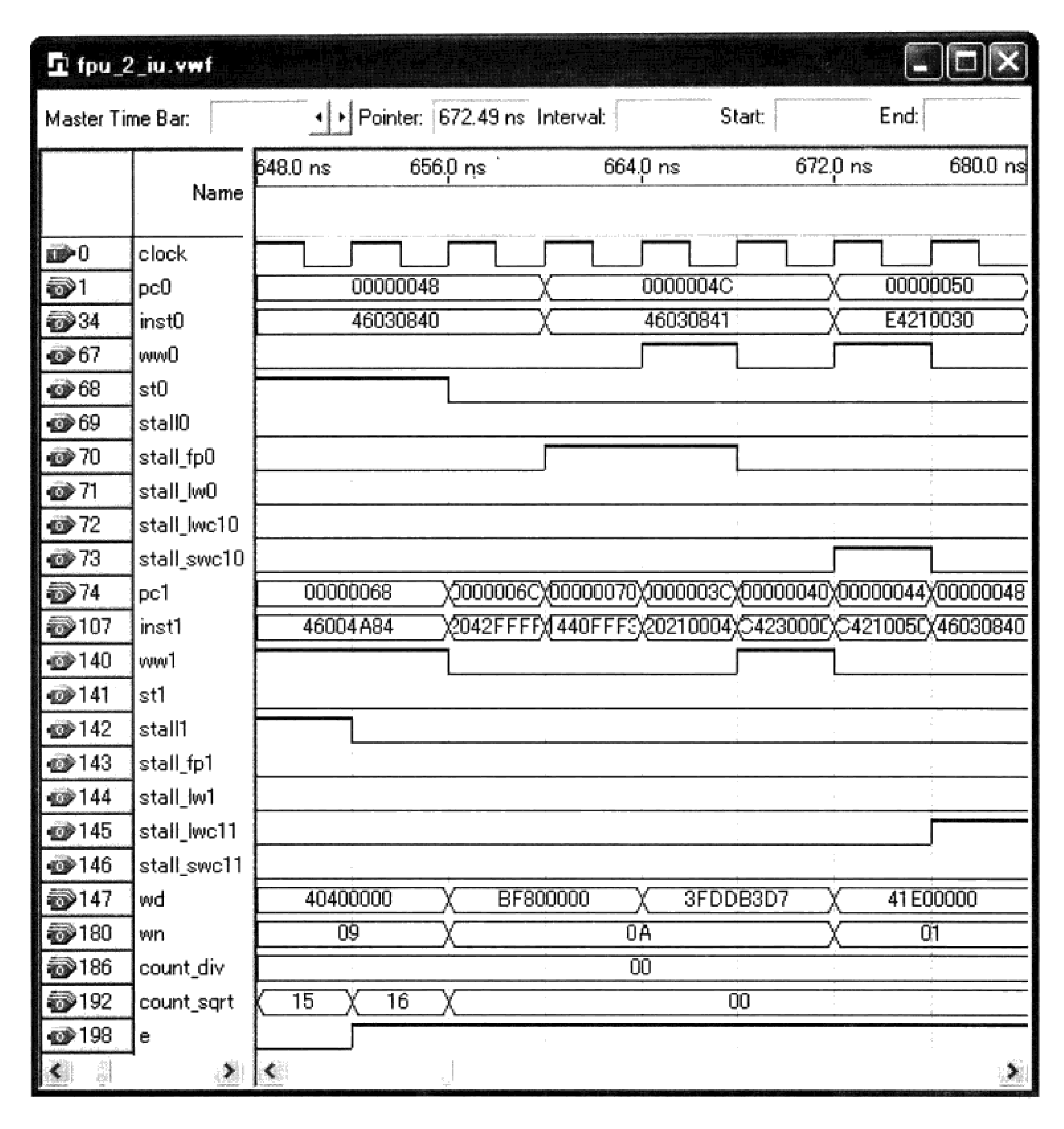
图 11.11 示出的也是浮点部件连续执行需要迭代的指令 (除法指令或开方指令) 时的情况。图中的 276ns 处，stall1 变 0，导致使能信号 e 变 1，进而结束线程 1 的 div.s 指令的 ID 级。从 280ns 处开始，线程 0 进入 sqrt.s 指令的 ID 级，stall0 = 1。与此同时，线程 1 试图对 sqrt.s 指令进行译码，但这时线程 0 正处在 sqrt.s 指令的译码期间，线程 1 必须等待 (st1 = 1)。




```
17 : 460629C3; % (5C)    div.s  f7,  f5, f6  # div      %
18 : 46004244; % (60)    sqrt.s f9,  f8      # sqrt     %
19 : 46004A84; % (64)    sqrt.s f10, f9      # sqrt     %
```

图 11.12 多线程 CPU 仿真波形图 (保存浮点除法结果、线程 0 浮点开方开始)

图 11.12 示出的是线程 1 完成第一条 sqrt.s 的译码，然后线程 0 开始对第二条 sqrt.s 指令进行译码的波形。线程 1 也试图对第二条 sqrt.s 指令进行译码，但由于资源冲突，它必须等待 (st1 = 1)。由于第二条 sqrt.s 指令与第一条 sqrt.s 数据相关，因此 stall_fp0 = 1、stall_fp1 = 1。从 464ns 开始，线程 0 和线程 1 相继把 div.s 指令的执行结果 (40000000) 写入各自的浮点寄存器 f7。




```
19 : 46004A84; % (64)    sqrt.s f10, f9       # sqrt          %
1A : 2042FFFF; % (68)    addi   r2,  r2, -1   # counter - 1   %
1B : 1440FFF3; % (6C)    bne    r2,  r0, 13   # finish?       %
```

图 11.13　多线程 CPU 仿真波形图 (保存浮点开方结果)

图 11.13 示出的是线程 1 保存第二条 sqrt.s 指令的执行结果前后的波形。当开方计数器值为十六进制的 16 时，ID 级将结束，然后经过 E1、E2、E3，线程 1 在 WB 级把浮点开方结果 (3FDDB3D7) 写入浮点寄存器 f10。接下来的两个线程的指令都不是浮点除法或浮点开方指令，因此两个计数器的值均为 0、使能信号 e 为 1。其他指令的波形比较简单，就不列出了。

11.4 习题

1. 重新设计线程选择电路，使得线程 0 被执行的概率高于线程 1 被执行的概率，比如 (5/8)：(3/8)。
2. 重新设计 FPU 和线程控制部件：将 FPU 的加减、乘、除和开方分开：若各线程的操作不同，则可以并行执行，比如一个线程在做除法，而同时另一个线程在做开方。
3. 试设计一个 4 线程的 CPU。

第 12 章　存储器和虚拟存储器管理

总的来讲，存储器 (Memory)，特别是随机访问存储器 RAM (Random Access Memory)，是保存程序 (指令和数据) 的临时场所。当一个程序要投入运行时，操作系统把这个程序整个或部分地从硬盘调入到存储器，然后交由 CPU 去执行。CPU 执行程序时，要从存储器中取指令。存储器访问指令，比如 lw 或 sw，还要从存储器取数据或把数据写入存储器。因为存储器的速度比 CPU 慢很多，在 CPU 内部都设计有高速缓冲存储器 (以下简称 Cache)。

另外，所有编译好的程序都使用它自己的虚拟地址空间。一般来讲，虚拟地址空间都是从 0 开始。如果访问存储器时直接使用这个虚拟地址，则会造成不同的程序之间相互冲突。因此，当一个程序被调入到存储器时，还需要根据当前存储器的使用情况，为它安排空闲的存储器。这就需要有一个机制，把程序执行时的虚拟地址转换成实际的存储器地址。为了加速地址转换，CPU 内部通常设计有 TLB (Translation Lookaside Buffer)，它有与 Cache 非常类似的结构。

本章主要讨论各种存储器的原理、Cache 的结构与设计、虚拟存储器管理、用于快速地址转换的 TLB 以及 MIPS CPU 的基于 TLB 的地址转换机制。

12.1　存储器

我们知道，存储器是构成计算机的三剑客之一 (另外的两剑客是 CPU 和 I/O 接口)。存储器种类繁多，依用途或内部结构不同，我们把它大致分成以下 4 类。

1) 静态存储器 (Static Random Access Memory，SRAM)，主要用于 Cache 和 TLB 设计，有钱人用它来实现计算机主存；
2) 动态存储器 (Dynamic Random Access Memory，DRAM)，用于实现主存；
3) 只读存储器 (Read-Only Memory，ROM)，用于存放初始启动程序 (固化)；
4) 相联存储器 (Content Addressable Memory，CAM)，用于 Cache 和 TLB 设计。

除了 ROM，其他三种存储器都具有所谓的“挥发性”，意即电源关掉后，原来保存在存储器中的内容便消失得无影无踪。因此，刚开机时，RAM 中的内容是不可使用的。那么，开机时 CPU 执行的第一条指令从何而来？答案是从 ROM 中来。本节简要地描述这 4 种存储器的原理和结构。

12.1.1　静态存储器 (SRAM)

静态存储器是相对于动态存储器而言的，或者说动态存储器是相对于静态存储器而言的。简单地讲，动态存储器需要“输氧”，也就是所谓的刷新 (Refresh)。否则的话，动态存储器中的内容会渐渐消失。而静态存储器很“健康”，不需要刷新。大家应该还记得 D 锁存器吧。静态存储器的一位就类似于一个 D 触发器。

虽然没有谁这么干，但还是让我们来看看如何用 D 触发器来构成一个小容量的静态存储器[26]。图 12.1 示出的是用 12 个 D 触发器设计的一个存储器模块。它一共有 4 个字，每个字 3 位。地址是 A[1:0]、数据输入端是 DI[2:0]、输出端是 DO[2:0]、WE 是写使能。数据输入端接到 D 触发器的 D 端。左侧是地址译码，用于选择 4 个字中的一个。译码的输出和 WE 相与 (AND)，接到 D 触发器的允许端 C。D 触发器的输出 Q 和地址译码的输出相与再送到一个或门，形成数据输出信号。

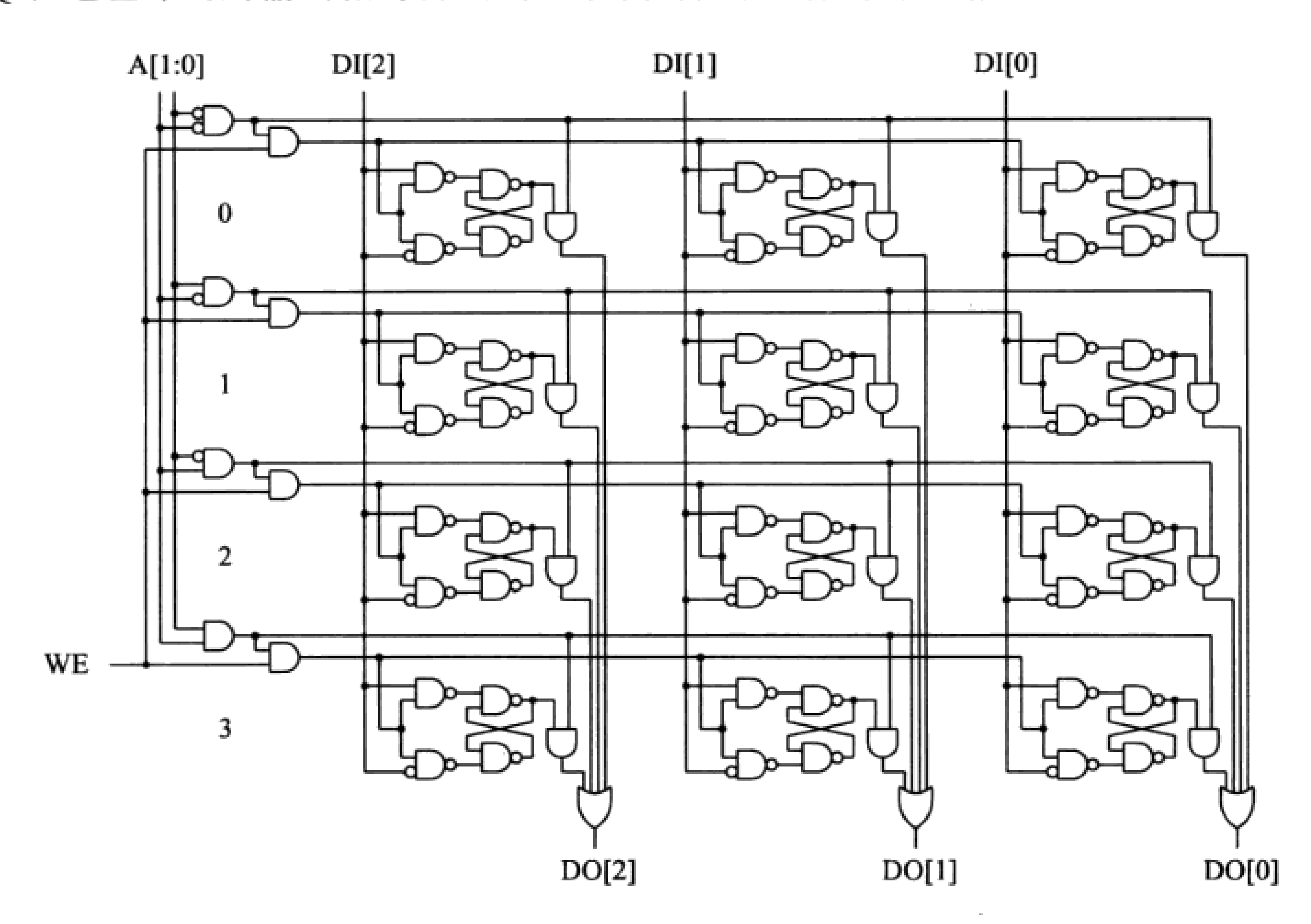


图 12.1 由 D 触发器构成的 4 × 3 位静态存储器 (供演示用)

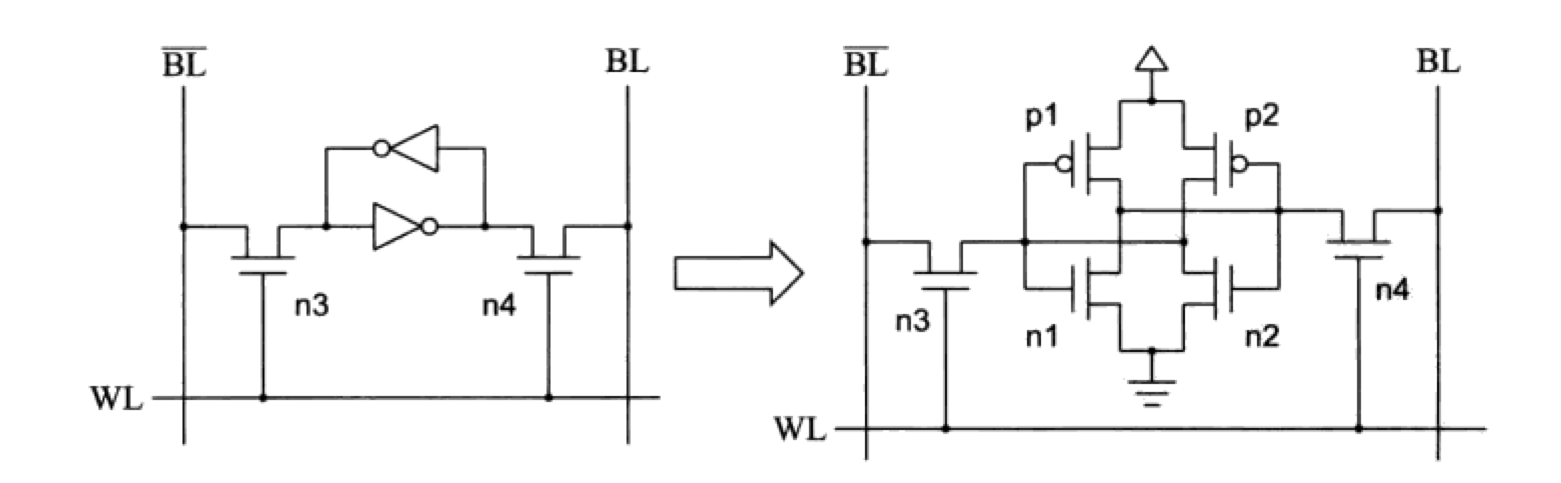


图 12.2 一种 6 晶体管静态存储器 SRAM 的存储单元

实际的静态存储器单元不是使用 D 触发器，而是使用类似于图 12.2 所示的电路。左侧是简化的电路图。两个非门构成一个双稳态的存储单元；BL 和 $\overline{\mathrm{BL}}$ 是数据线；WL 是一个字的选择信号，相当于 12.1 的地址译码输出。它控制 n3 和 n4 两个

晶体管，当它为高电平时，可对存储单元进行读写。右侧是详细的 CMOS 电路，p1 和 n1 组成一个非门；p2 和 n2 组成另一个非门。

静态存储器的特点是与 CPU 的接口简单且速度快，但价格高，耗电量也大。因此一般用于 Cache 和 TLB 设计，但有一些高性能计算机也拿它当主存用。

12.1.2 动态存储器 (DRAM)

与静态存储器的存储单元不同，动态存储器使用一个小容量的电容来保存信息，用电容中有无电荷 (电平的高低) 来表示 1 和 0。图 12.3 是一个 1M × 1 位 DRAM 芯片的内部结构示意图。“×1”是指数据线外部接口的位数。1M 个单元 (位) 组成 1K × 1K 的二维阵列。10 位行地址译码器的输出用于选择阵列的一行 (1K 位)。列地址译码器 / 多路选择器从一行的 1K 位中选择一位。访问 1M 个单元需要 20 位地址，但一般的 DRAM 芯片的地址线只有所需地址位数的一半。因此地址要分两次送给 DRAM 芯片，低电平有效的 RAS (Row Address Strobe) 和 CAS (Column Address Strobe) 分别用于选通行地址和列地址。

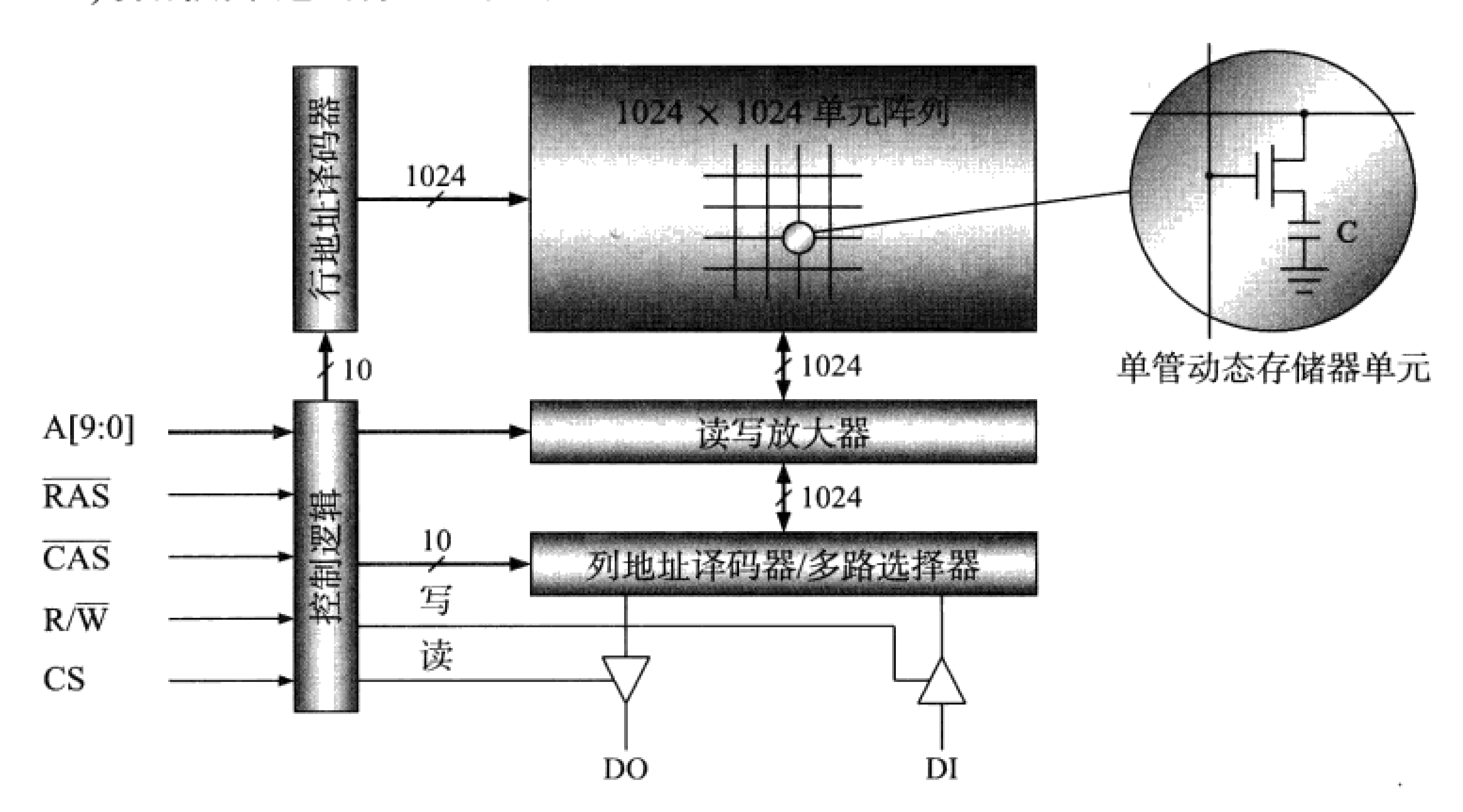


图 12.3 动态存储器 1M × 1 位 DRAM

图中的一位数据线有单独的输入和输出管脚 (DI 和 DO)，也有共用一个管脚的 (双向数据线)。数据位数也有多于一位的。另外，电容会漏电，所以 DRAM 需要定期刷新。刷新不是一位一位地进行，而是一行一行地进行。

12.1.3 只读存储器 (ROM)

只读存储器也是可以随机访问的。与 RAM 不同，ROM 存储器的内容即使断了电也不会丢失。计算机系统中都有 ROM，用于存放操作系统的初始引导程序等。

ROM 有很多种。常见的有 MROM (Mask ROM)、PROM (Programmable ROM)、EPROM (Erasable PROM)、EEPROM (Electrically EPROM) 和 Flash Memory 等。Flash Memory 与一般的存储器不同，它不支持传统意义上的随机访问。与其说 Flash Memory 是一种存储器，倒不如说是一种类似于硬盘的外部存储设备。

12.1.4 相联存储器 (CAM)

还有一种比较特殊的存储器 (Content Addressable Memory，CAM)，按英文字面可译成“可按内容寻址的存储器”，但读起来不怎么顺。本书称其为相联存储器。不管名称如何，关键是要看它的内涵。

传统存储器的读操作是：给个存储器地址，相应的存储器数据就出来了。对相联存储器可以这样简单地理解：它与传统存储器刚好相反，给个存储器数据，该数据在存储器中的位置 (地址) 就出来了。当然，如果那个数据不在存储器中，也不可能有地址出来。即，相联存储器查找存储器中所有的内容，看看是否有一个或多个与输入数据匹配的单元。如果有，在哪儿？

图 12.4 是一个 4 字 3 位的相联存储器 CAM 的结构示意图。12 个 C 是 12 位存储器数据，3 位输入数据的每一位都有正反两个信号，SL_i 和 $\overline{SL_i}$、$0 \leqslant i \leqslant 2$，接到所有字的相应的 C 位 (列)。SL 是 Search Line 的缩写。当某个字的 3 位 (行) 均与输入数据相同时，其 ML 信号为高电平。ML 是 Match Line 的缩写。如果有多个字匹配，那么就有多条 ML 线输出高电平。右边是一个优先级编码器，从中选出一个地址 Matched_Address。信号 Match_Found 表示是否有匹配。在有些电路的设计中可能并不需要这个编码器，而是直接输出全部的 ML。

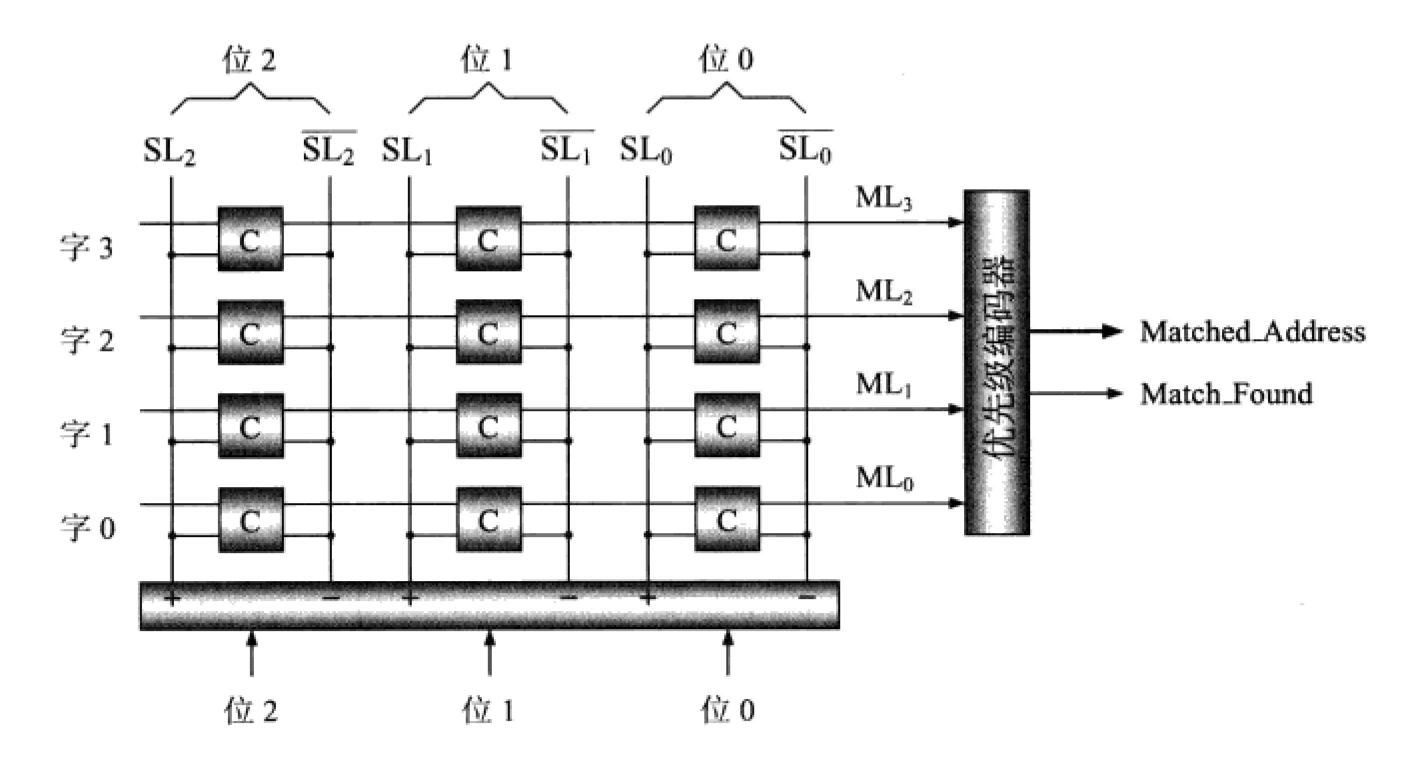


图 12.4 相联存储器 CAM 的结构图

那么存储单元 C 到底是一种怎样的结构，这么厉害，能实现匹配呢？这样的电路有许多种实现方法[25]，图 12.5 给出的是一种非常朴素的电路。图中的存储单元部

分与 SRAM (见图 12.2) 相同，它的输出用 D 和 $\overline{\mathrm{D}}$ 表示。不匹配时，$\overline{\mathrm{D}}$ 与 SL 相同，D 与 $\overline{\mathrm{SL}}$ 也相同，两组晶体管 (n1 和 n2 一组、n3 和 n4 一组) 总有一组导通，导致 ML 变低。反之，每组中都有一个晶体管截止，维持 ML 的高电平 (匹配)。ML 连接到多位这样的存储单元，只要有一位不匹配，ML 就被拉低。与图 12.4 相比，图 12.5 中多出了 WL，它和 SRAM 中的 WL 相同，用于 SL 写入时的行选择。

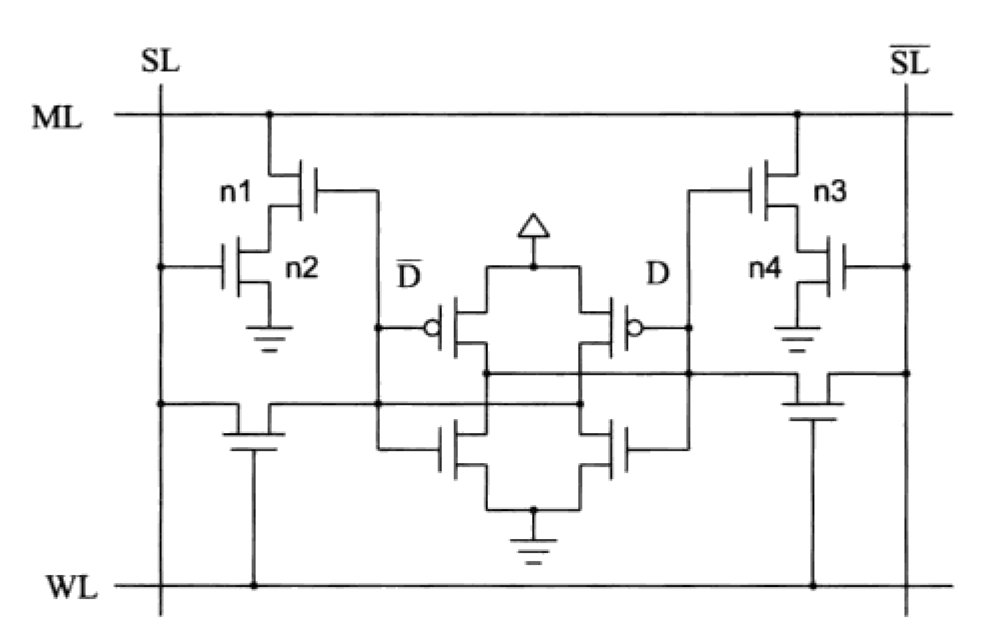


图 12.5 一种相联存储器 CAM 的存储单元

相联存储器可以用于各种有按位匹配要求的场合，比如虚拟存储器管理、数据压缩、Cache、TCP/IP 中的 IP 地址匹配等。图 12.6 示出的是相联存储器的应用之一：Cache 的简单结构图。图中的左侧用存储器地址去匹配 CAM 相联存储器。如果有匹配发现，其匹配地址被用来访问快速的 RAM，从中得到指令或数据。图 12.6 是使用独立的 CAM 和 RAM 芯片设计 Cache 的例子。如果把它们集成在一个芯片中，则可以省去编码器和译码器，把 CAM 匹配线直接和 RAM 行选择线相连。但这时不允许有多个匹配同时出现。

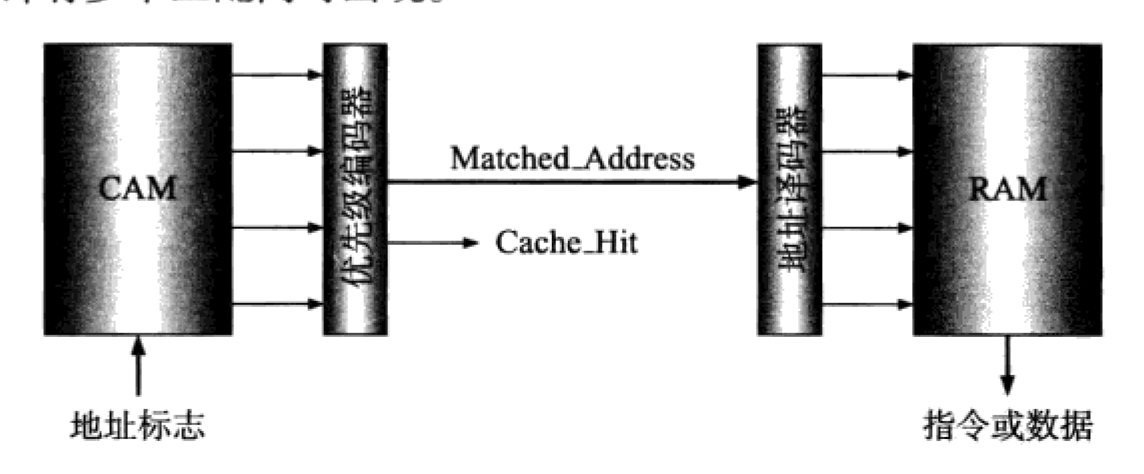


图 12.6 使用相联存储器的 Cache 示意图

12.1.5 存储层次

图 12.7 示出了目前计算机系统典型的存储层次结构。从左至右，存储容量增大，但速度变慢。使用这样一个存储层次的目的是为程序提供一个容量大、速度快且价格低的存储系统。

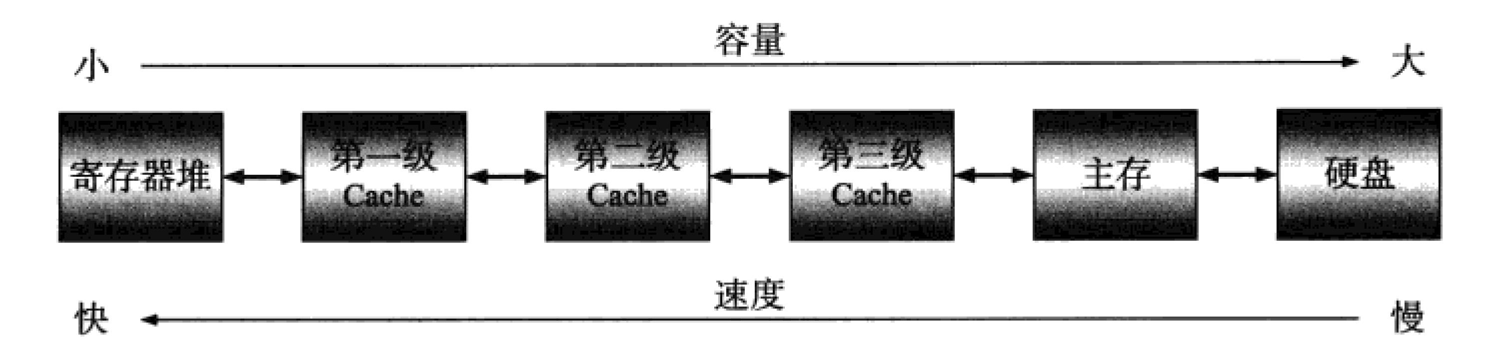


图 12.7 存储层次

寄存器堆是速度最快的存储部件，它在 CPU 芯片内部。CPU 使用指令中的寄存器号直接访问它。一般的 CPU 有 16 ~ 256 个寄存器。Cache 存放主存中的部分数据。第一级 Cache 和第二级 Cache 通常也在 CPU 片内 (On-Chip Cache)，而且第一级有分开的指令 Cache 和数据 Cache。第三级 Cache 在片外，使用 SRAM。Cache 对用户程序来讲是透明的，即用户看不到它，没有办法用程序对它进行控制或访问。在有些计算机的体系结构中，Cache 对操作系统也是透明的；而有些体系结构则提供了特权指令对 Cache 直接进行维护。主存 (Main Memory) 是用户能访问的存储器，存放指令和数据。硬盘的主要功能是存放文件，但它的另一功能也很重要，即与主存一起为用户提供一个相对大的虚拟存储器空间。如果我们把主存称为第一级存储器，则硬盘是第二级存储器。

12.2 高速缓存 (Cache)

Cache 的原意是一个隐蔽的场所，用来保存食物等贵重的东西。在计算机系统中，借用 Cache 来表示一个小容量高速度的缓冲区，用于存放 CPU 经常使用的原本在主存中的指令或数据以加快 CPU 访问存储器的速度。

前面我们已经讲过，Cache 对用户程序来讲是透明的，即用户看不到它。为什么呢？因为 Cache 是一个隐蔽的场所，让用户看到它就谈不上隐蔽了。实际的意义是：对 Cache 的管理是我们硬件的内政，不容你们软件干涉。

最初，Cache 中什么也没有。所以 CPU 必须访问主存。从主存拿到的指令或数据由硬件写入 Cache 中，以便 CPU 以后再访问相同的地址时，不用访问主存，直接从 Cache 中得到。

假设 Cache 的内容写入后就再也没被 CPU 使用过，那么这个 Cache 就没有任何正面的意义。使用 Cache 是基于指令和数据访问的局部性 (Locality) 特性。局部性分空间局部性 (Spatial Locality) 和时间局部性 (Temporal Locality)。空间局部性是指当 CPU 访问某个存储单元时，该存储单元附近的存储单元最有可能被随后访问；时间局部性是指 CPU 访问某个存储单元后，该存储单元最有可能被再次访问。

如果 CPU 想要的指令或数据在 Cache 中找到了，我们说 Cache 命中 (Hit)。如果没命中 (Miss)，CPU 必须要访问主存。假设 Cache 的访问时间 $t_c = 1$ns、主存的访问时间 $t_m = 50$ns、Cache 的命中率 $h = 98\%$，则平均的存储器访问时间是

$$
\begin{aligned}
t &= h \times t_c + (1-h) \times (t_c + t_m) \\
&= t_c + (1-h) \times t_m \\
&= 1 + 0.02 \times 50 = 1 + 1 = 2ns
\end{aligned}
$$

Cache 的容量比主存小得多。那么主存的指令或数据放在 Cache 的什么地方？再者，当 Cache 已满，“新来的”要替换掉谁？如果往存储器写入数据时 Cache 没命中怎么办？以下各小节回答这些问题。

12.2.1 Cache 的映像机制

Cache 映像机制定义数据在 Cache 中存放的规则。主要的映像机制有三种：

1) 直接映像 (Direct Mapping)；
2) 全相联映像 (Fully Associative Mapping)；
3) 组相联映像 (Set Associative Mapping)。

1. 直接映像

直接映像使用地址的低位作为 Cache 存储器的地址直接访问 Cache。那么地址的高位怎么办？不能简单地忽略，否则会把不是属于你的东西拿来了。因此我们要在 Cache 中保存自己的高位地址作为一个标志 (Tag)。考虑到存储器访问的空间局部性的特点以及减少标志所占的存储单元的数量，Cache 通常是以块 (Block) 或行 (Line) 为单位与主存之间交换数据。这样，每个 Cache 块除了有数据，还要配备一个标志 Tag (地址的高位)。如果把低位地址看作“名”，那么标志就是“姓”，姓和名合起来就是姓名，能确定 Cache 中的东西到底是谁的。

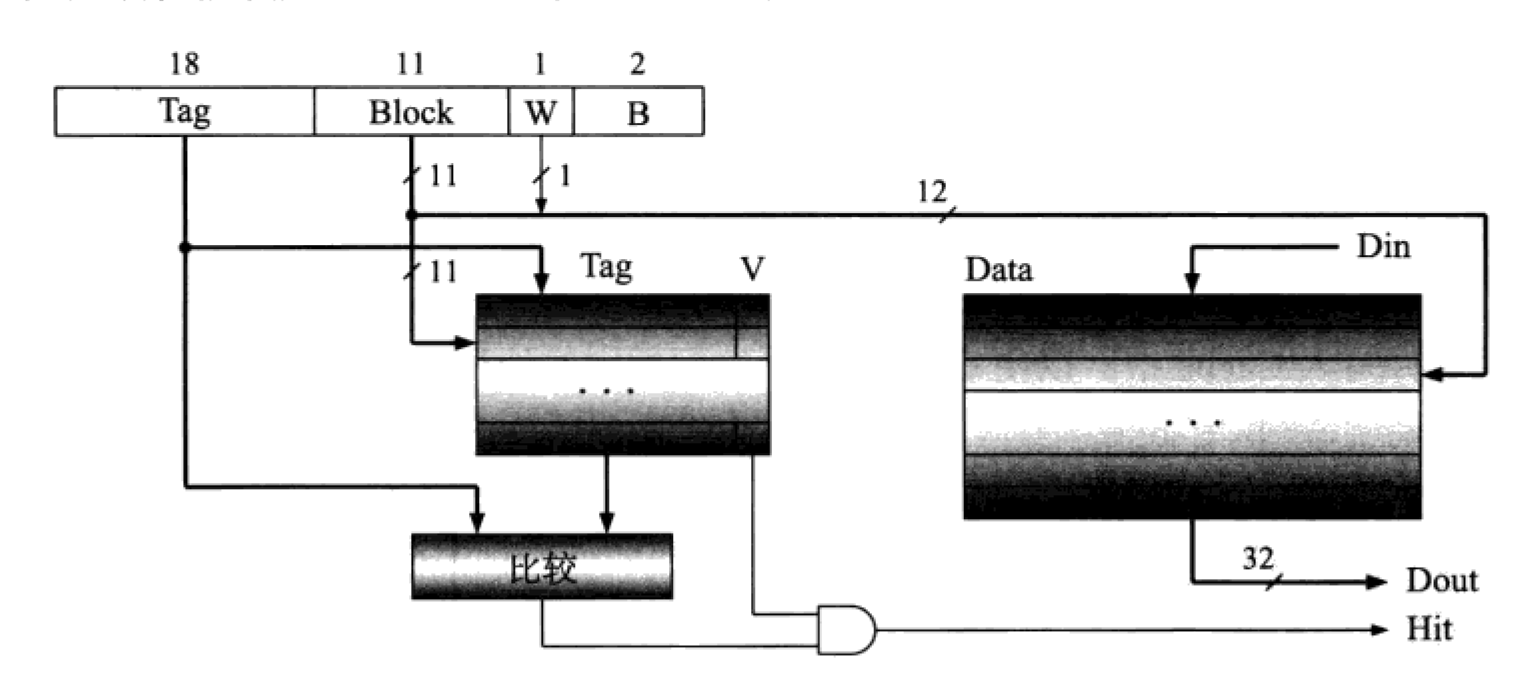


图 12.8 直接映像 Cache

图 12.8 给出的是 16KB (16×2^{10} 字节) 直接映像 Cache 的硬件结构。图中的地址是 32 位，其中的 Block 是 Cache 块号，或称块地址，W 是块内的字地址。一个字

有 4 个字节，B 是字节地址。由于 Cache 有 16KB = 2^{14} 字节，需要地址的低 14 位作为 Cache 的地址。从图中可以看出，每块为 8 = 2^3 个字节，或两个字。因此访问 Cache Tag 部分的地址为 14 − 3 = 11 位 (块地址)。而访问 Cache Data 部分的地址为 12 位 (字地址)，输出的数据是 32 位 (一个字)。另外图中的 V (Valid) 是相应块的有效位。当地址中的标志与 Cache 中的标志相等、并且有效位也为 1，我们说 Cache 命中 (Hit = 1)。注意，图中数据部分的 Cache 存储器可以直接使用地址中的 W 位，而不用在外面加一个二选一多路器来选择一块 (两个字) 中的一个字。

2. 全相联映像

图 12.9 示出的是 16KB 全相联映像 Cache 的硬件结构。直接映像 Cache 的特点是存储器的数据在 Cache 中存放的地点是唯一的，一点迁徙的“自由”也没有。与此相对照，全相联映像 Cache 有最高的“自由度”，愿意住哪儿就住哪儿。多么和谐的社会啊。但直接映像 Cache 使用普通的静态存储器 SRAM 就行，而全相联映像 Cache 需要使用相联存储器 CAM。看来要自由是要付出代价的。

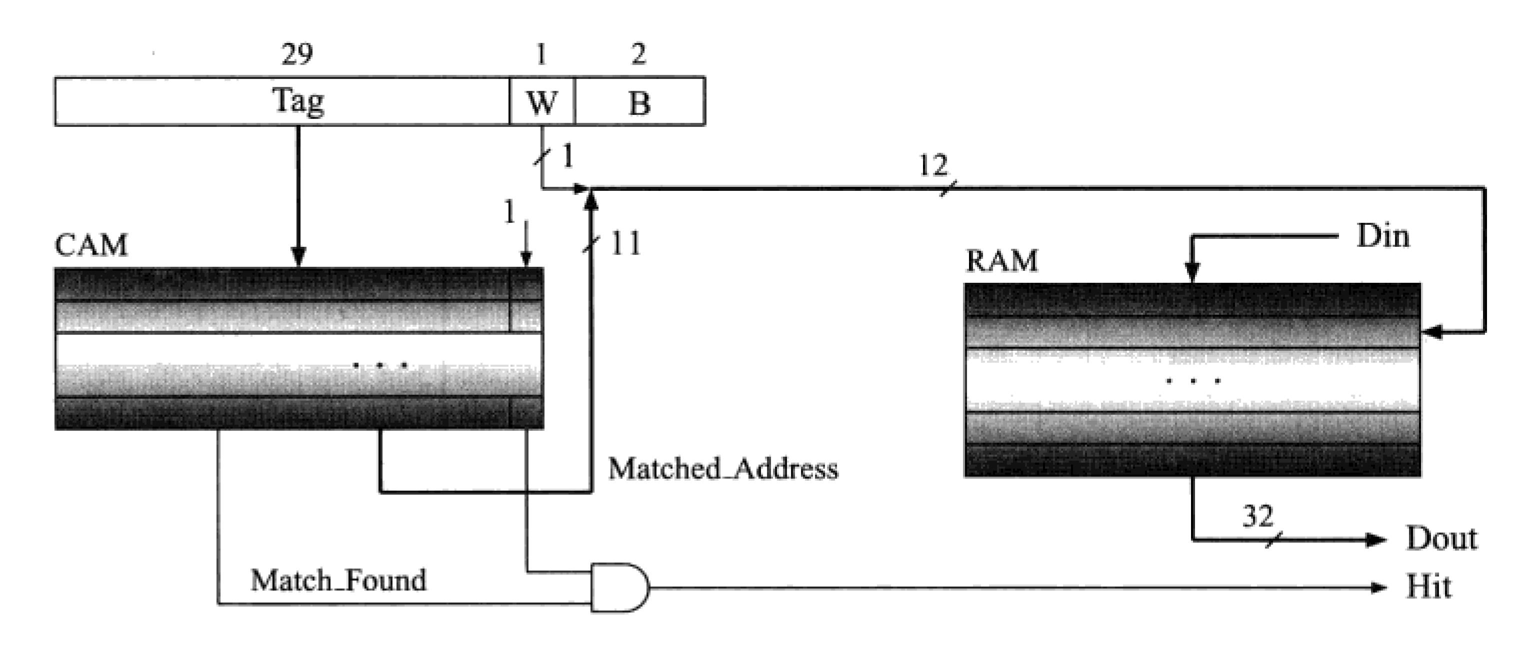


图 12.9　全相联映像 Cache

图中地址标志的比较电路使用 CAM。有关 CAM 的原理及内部单元的 CMOS 电路我们已经在本章开始处介绍了。若标志匹配，CAM 输出 11 位的匹配地址 Matched_Address，它与地址中的一位 W 一起，构成地址访问 Cache 数据部分的存储器 RAM。如前所述，CAM 和 RAM 可以集成在一起，省去地址编码器和地址译码器，用 CAM 匹配线直接选中 RAM 的一行 (块)，再用 W 选出一个字。

3. 组相联映像

在“自由度”这个问题上，我们还是可以商量的。直接映像和全相联映像是两个极端，而组相联映像在二者之间搞“中庸”。尽管同样是中庸，但也有偏向谁的问题，其衡量指标是“路”(Way) 的多少。二路组相联是把直接映像的 Cache 分成两路，每路的规模是总数的一半。存储器数据存入 Cache 时，用地址的低位对两路同

时访问，看看有没有一路空闲。如果有，先住进去再说。读 Cache 时也是一样，用地址的低位对两路同时访问，看看有没有一路命中。

二路组相联是最偏向直接映像的“中庸”，也就是随便应付一下。如果你觉得怠慢了全相联映像，你可以用“四路”、“八路”、“十六路”等。当路数增至与 Cache 总块数相同，就变成全相联映像了。你看，由量变到质变了不是。

图 12.10 示出的是 16KB 二路组相联映像 Cache 的硬件结构。Tag 部分一半一半分成两路，使用块地址同时访问这两路，因此块地址只要 10 位就够了 (比直接映像少了一位，但标志多了一位)。同时被选中的两路合在一起称为一组 (Set)。组内可以任意存放，即，选择一组时用直接映像，组内用全相联。访问 Cache 的数据部分仍用 12 位地址，其中 10 位来自于块地址 Block，另两位分别来自于 W 和 Hit1。

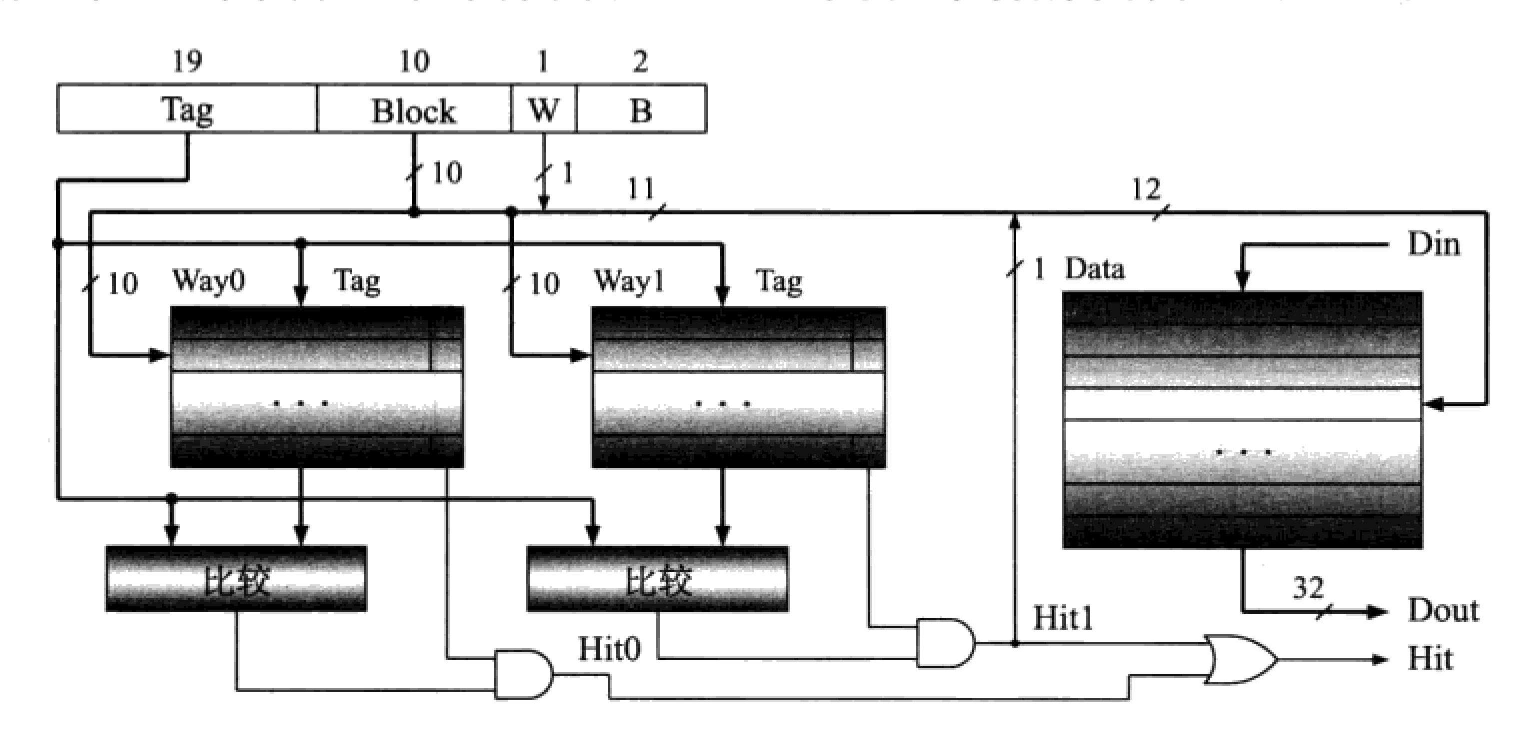


图 12.10 二路组相联映像 Cache

除了以上讲的三种映像，还有一些不太常用的其他映像方式，比如“扇区映像”(Sector Mapping)。映像方式不同，直接导致 Cache 的性能不同。我们对 Cache 的性能进行评价时，不但要看它们的命中率，也要看价格及访问速度。

12.2.2 Cache 块的替换算法

以全相联 Cache 为例。当一块数据被存入 Cache 时，首先查找有没有空闲的 Cache 块。如果有，就把数据存入该块。如果全相联 Cache 所有的块均被存入了数据，那么再有新的块又要存入 Cache 时怎么办呢？答案是只好把原来已在 Cache 中的块替换掉了。替换谁呢？这倒是一个伤脑筋的问题。同样，对于组相联 Cache 来讲，由于组内是全相联，也有替换掉哪一路的问题。直接映像的 Cache 有没有这个问题呢？没有，因为是直接映像，没有其他的选择。所以直接映像的 Cache 不用伤这个脑筋。

为了提高 Cache 的命中率，我们希望把“将来”不再被使用或很久很久以后才被使用的 Cache 块替换掉，而把那些近期将要被使用的块保留。但是，希望归希

望，“将来”的事情谁也说不准。

使用 Cache 的目的是缩短存储器的平均访问时间，因此 Cache 块的替换算法一般由硬件实现。以下我们介绍两种相对简单的替换策略：LRU 算法和随机算法。

1. LRU 替换算法

LRU (Least Recently Used) 是使用最为广泛的一种替换算法。LRU 的意思是“最近最少被使用”，即 LRU 替换算法不是“展望未来”，而是“回顾过去”。就像新招一个队员同时让一个“板凳队员”卷铺盖走人，实现 LRU 算法要求对每个“队员”做记录，看谁在板凳上坐的时间最长。

以组相联映像 Cache 为例。假设一组有八路，则每路要有一个三位计数器，通过各路的计数值来区分哪一路最近老没被教练派上场。规则如下：

1) 替换掉计数值为 0 的块 (如果有多于一个块的计数值为 0，随便替换哪一个)。设置该块的计数器为最大值 7，其他块的计数器都减 1。但若计数值是 0 就不再减了。我们称这样的计数器为饱和计数器 (Saturated Counter)。
2) 命中时，假设命中块的计数值为 k，则把计数值大于 k 的所有其他块的计数器减 1，并把命中块的计数器设置为最大值 7。
3) 计算机刚启动时，把所有的 Cache 块的计数器和有效位全部清零。这意味着该队连一个队员也还没有呢。

如果一组有四路，则需为每路安排一个两位计数器。如果一组只有两路，按推理需为每路安排一个一位计数器。但是，如果其中的一路是最近最少被使用的话，不言而喻，另一路一定是最近最多被使用的，因此只用一个一位计数器就足够了。

以上讨论的是一组中的情况。如果 Cache 总的组数为 S，则总的计数器的数量是一组中计数器的个数乘以 S。这是一笔不小的开销。

2. 随机替换算法

LRU 算法是通过总结历史经验来预测将来的发展方向。这种算法需要“投资”，即为每一块配备一个计数器。

如果你想省钱，随机 (Random) 算法倒是一个相当不错的选择。随机算法完全不管 Cache 块过去、现在及将来的使用情况，简单地根据一个随机数，选择一块替换掉。注意，整个 Cache 只需一个随机数产生器，所以比 LRU 省钱。随机数可由硬件产生，例如设置一个计数器，由系统时钟进行计数。是不是被替换掉就看你的运气了。虽然有些 Cache 设计者讨厌这种方法，但模拟结果证明，它的性能还是相当不错的，至少把钱先省了。

还有一种先进先出 FIFO (First-In First-Out) 算法：不管你在场上的表现如何，谁先入队谁先走人。与 LRU 算法一样，也需投资。虽然还有其他一些替换算法，但实现起来过于复杂。在大多数 Cache 设计中，二路和四路组相联映像 Cache 用 LRU 替换算法，其他的用随机替换算法 (直接映像 Cache 用不着替换算法)。

12.2.3 Cache 写策略

截至目前，我们都是在讨论读存储器时如何对 Cache 进行管理。那么写存储器时又是怎样的一种情况呢？看以下的一些 Cache 写策略。

1. 写透和写回

写存储器并且 Cache 命中时，有以下两种策略可供选择：写透 (Write Through) 和写回 (Write Back 或 Copy Back)。写透策略是指在写 Cache 的同时也写主存，也可译成写穿、写通、通写或统写 (统统写)，总之是 Cache 和主存 “通吃”。它的优点是能够保持主存与 Cache 的一致性；缺点是增加数据传输量，并且写存储器要花费较长的时间 (可以把数据先放在快速的写缓冲区内，然后再由硬件慢慢写)。

写回策略是只写 Cache，不写主存。只有当数据块要被替换掉时，才将它写回主存。如果被替换掉的 Cache 块从来没被写过，即它的内容与主存相应块的内容是一样的，就不必写回主存了。因此，为了能够区分 Cache 块是否被写过，我们需要为每一块增加一位 “修改位” (Updated Bit 或 Dirty Bit)。

当数据块首次被调入 Cache 时，清除修改位。一旦往数据块写数据时，把修改位置 1。在数据块被替换掉时，若它的修改位为 0，则简单地把新的数据写入该块；如果修改位是 1，则先要把数据块写回主存，然后再调入新的数据块。这种策略的优点是缩短了写操作所用的时间，减少了存储器访问量；缺点是主存中可能会存有过时的数据。当其他 CPU 或 DMA 控制器从存储器中读数据时，有可能读不到最新的数据，即出现所谓的 Cache 一致性 (Cache Coherence) 问题。

2. 写前读入和写不读入

写存储器但 Cache 不命中时，也有两种写策略：写前读入 (Write Allocate) 和写不读入 (No Write Allocate 或 Write No Allocate)。写前读入策略是先把数据块从主存读入 Cache，然后再写。为什么不直接写 Cache 而是先读再写呢？因为 Cache 块是一 “大块”，要写入的只是一 “小块”，地址标志是为整个 “大块” 而准备的，因此必须把整块内容读进来。写不读入策略是绕开 Cache，只写主存。一般地，写前读入与写回策略一起使用，写不读入和写透策略又是一对儿。为什么呢？

12.2.4 数据 Cache 电路设计及 Verilog HDL 代码

图 12.11 是 Cache 与 CPU 及存储器之间的一般连接示意图。所有的信号分成两组：与 CPU 连接的信号名称以 p_ 开始；与存储器连接的信号名称以 m_ 开始。信号 a 是地址线。dout 和 din 是数据线。strobe 是选通线，为 1 时表示要进行读或写操作。rw 为 0 时表示读，为 1 写。ready 为 1 时表示已经准备好了：m_ready 表示存储器准备好了；p_ready 是通知 CPU：外面的世界准备好了。

图 12.12 是一个具体的 Cache 电路的实现：它使用直接映像方式及写透策略。图中的三个 RAM 模块分别存放有效位 (Valid RAM)、高位地址标志 (Tag RAM) 和

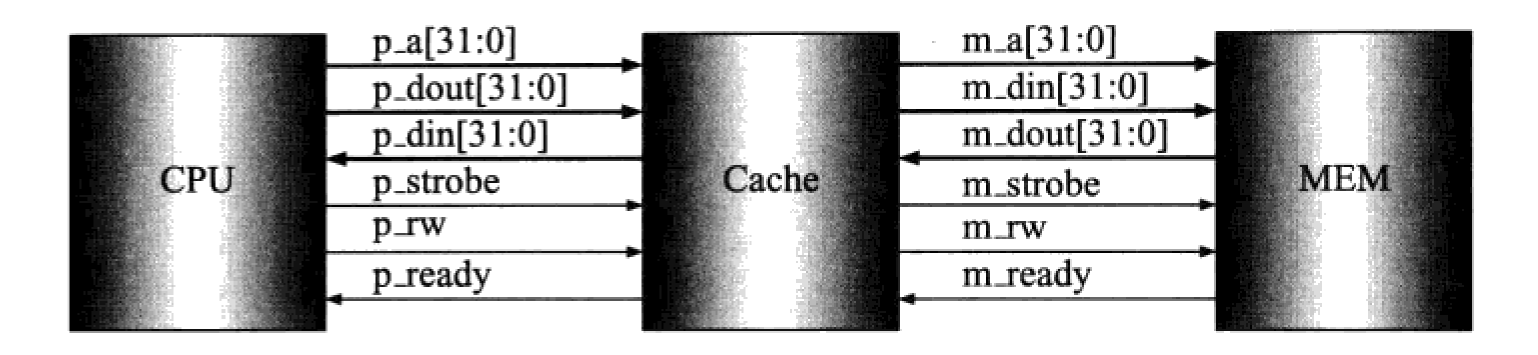


图 12.11 Cache 与 CPU 及存储器之间的接口信号

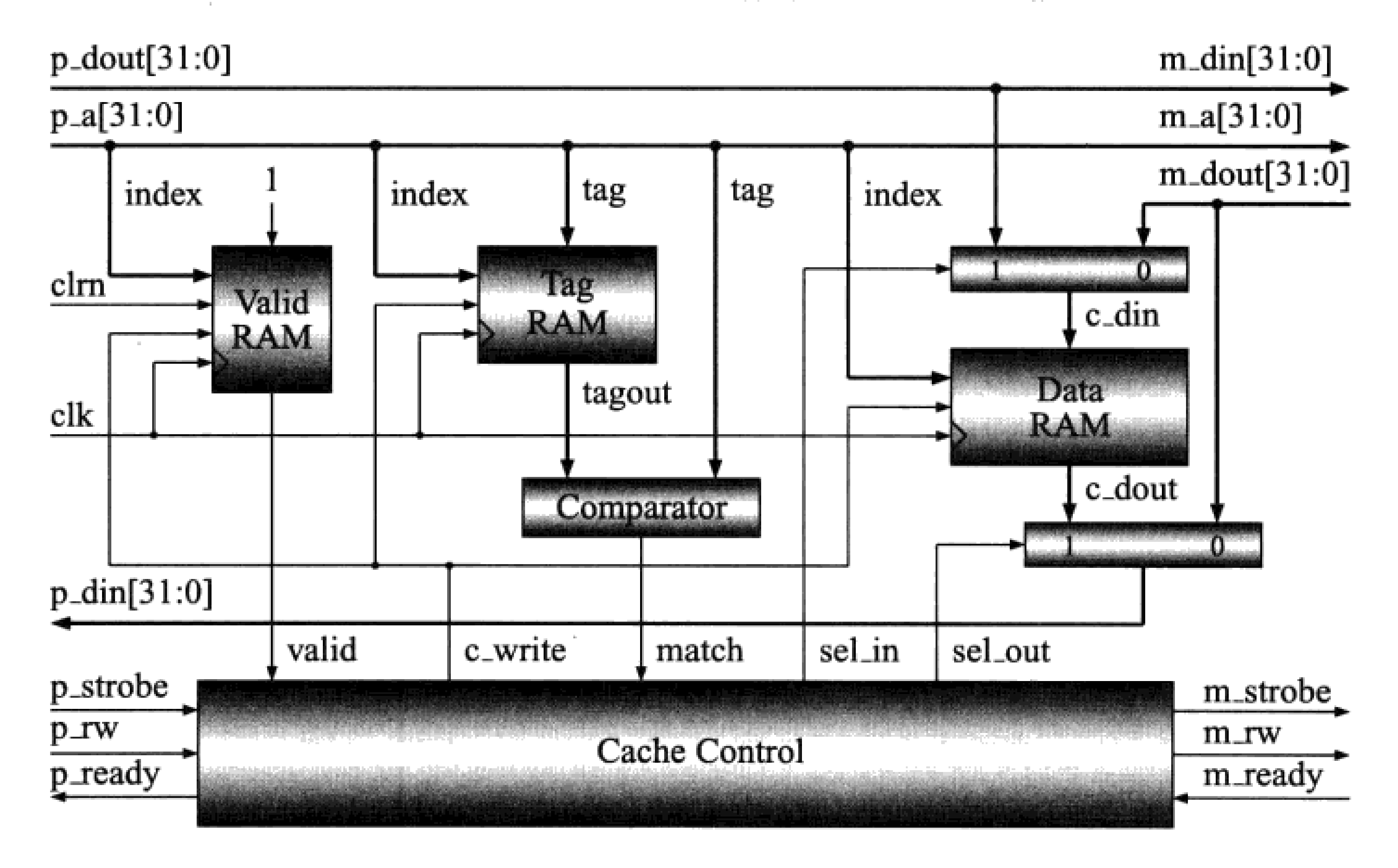


图 12.12 直接映像 Cache

Cache 数据 (Data RAM)，其中 Data RAM 的数据输入端和数据输出端各有一个二选一的多路器：写入 Cache 的数据有两个来源，一个是存储器、一个是 CPU。如果 Cache 没命中，从存储器取来的数据要写入 Cache 中。如果遇上存数据指令，则要把 CPU 的数据写入 Cache。送往 CPU 的数据 p_din 的来源也有两个，一个是 Cache 命中时的 Data RAM 数据，一个是没命中时从存储器取来的数据。两个多路器的选择信号由图中的控制电路产生。具体的实现见以下的 Verilog HDL 代码。

```
module d_cache #(parameter A_WIDTH = 32, parameter C_INDEX = 6)
  (p_a, p_dout, p_din, p_strobe, p_rw, p_ready, clk, clrn,
   m_a, m_dout, m_din, m_strobe, m_rw, m_ready);
    input [A_WIDTH-1:0]  p_a;
    input [31:0]         p_dout;
    output [31:0]        p_din;
    input                p_strobe;
    input                p_rw;      // 0: read, 1: write
```

```
output                 p_ready;
input                  clk, clrn;
output [A_WIDTH-1:0] m_a;
input  [31:0]          m_dout;
output [31:0]          m_din;
output                 m_strobe;
output                 m_rw;
input                  m_ready;
localparam T_WIDTH = A_WIDTH - C_INDEX - 2; // 1 block = 1 word
reg                    d_valid [0:(1<<C_INDEX)-1];
reg [T_WIDTH-1:0]      d_tags  [0:(1<<C_INDEX)-1];
reg [31:0]             d_data  [0:(1<<C_INDEX)-1];
wire [C_INDEX-1:0]     index = p_a[C_INDEX+1:2];
wire [T_WIDTH-1:0]     tag = p_a[A_WIDTH-1:C_INDEX+2];

// write to cache
always @ (posedge clk or negedge clrn)
    if (clrn == 0) begin
        integer i;
        for (i = 0; i < (1<<C_INDEX); i = i + 1)
            d_valid[i] <= 1'b0;
    end else if (c_write)
        d_valid[index] <= 1'b1;
always @ (posedge clk)
    if (c_write) begin
        d_tags[index] <= tag;
        d_data[index] <= c_din;
    end

// read from cache
wire                 valid = d_valid[index];
wire [T_WIDTH-1:0] tagout = d_tags[index];
wire [31:0]          c_dout = d_data[index];

// cache control
wire    cache_hit  = valid & (tagout == tag); // hit
wire    cache_miss = ~cache_hit;
assign m_din       = p_dout;
assign m_a         = p_a;
assign m_rw        = p_strobe & p_rw; // write through
assign m_strobe    = p_strobe & (p_rw | cache_miss);
assign p_ready     = ~p_rw & cache_hit |
                     (cache_miss | p_rw) & m_ready;
wire    c_write    = p_rw | cache_miss & m_ready;
wire    sel_in     = p_rw;
```

```
    wire    sel_out     = cache_hit;
    wire [31:0] c_din = sel_in ? p_dout : m_dout;
    assign      p_din = sel_out? c_dout : m_dout;
endmodule
```

存放有效位的 RAM 开始时要清零，其他两个不用。由于使用写透策略，每次执行存数据指令 (sw 或 swc1) 时都要写存储器，不管 Cache 命中与否。因此 p_ready 信号中包含了一项 p_rw & m_ready (存储器写并且存储器准备好)。如果 p_ready 为 0，CPU 要等待并维持存储器访问信号。另外，由于该例中一个 Cache 块只有一个字，不需要写前读入，因此写不命中时也写 Cache。

图 12.13 和图 12.14 是以上代码的仿真结果。图 12.13 示出的是读操作没命中以及命中时的部分波形。图 12.14 示出的是写操作以及读命中时的部分波形。

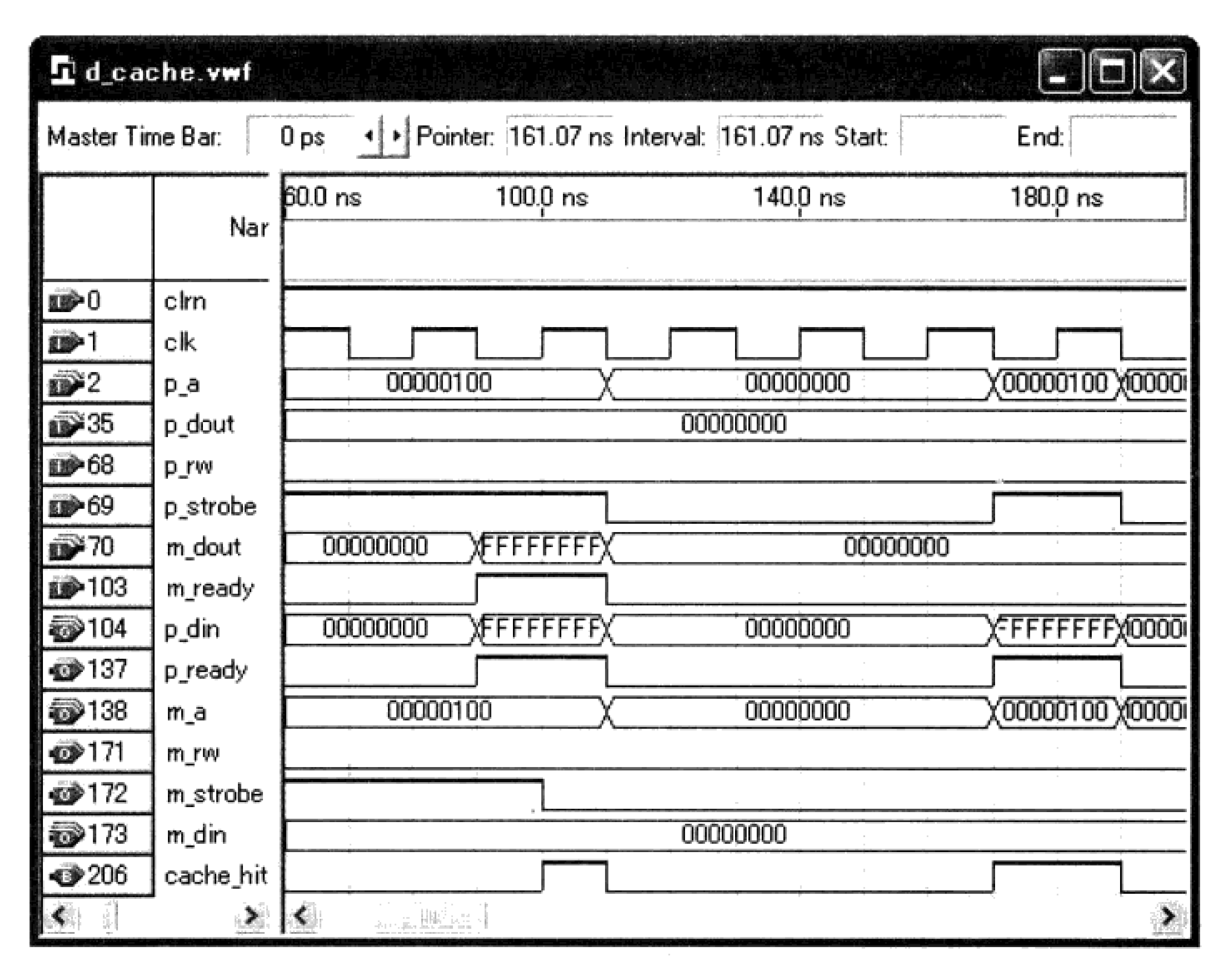


图 12.13 数据 Cache 仿真波形图 (读)

12.3 虚拟存储器管理及 TLB 设计

我们在本章开始处讲过，编译好的程序使用虚拟地址空间。当操作系统把程序调来执行时，为它分配存储器。在程序执行时，我们需要把程序中的虚拟地址转换成主存的实际地址。还有一些其他的名称，比如逻辑地址、处理机地址、程序地址

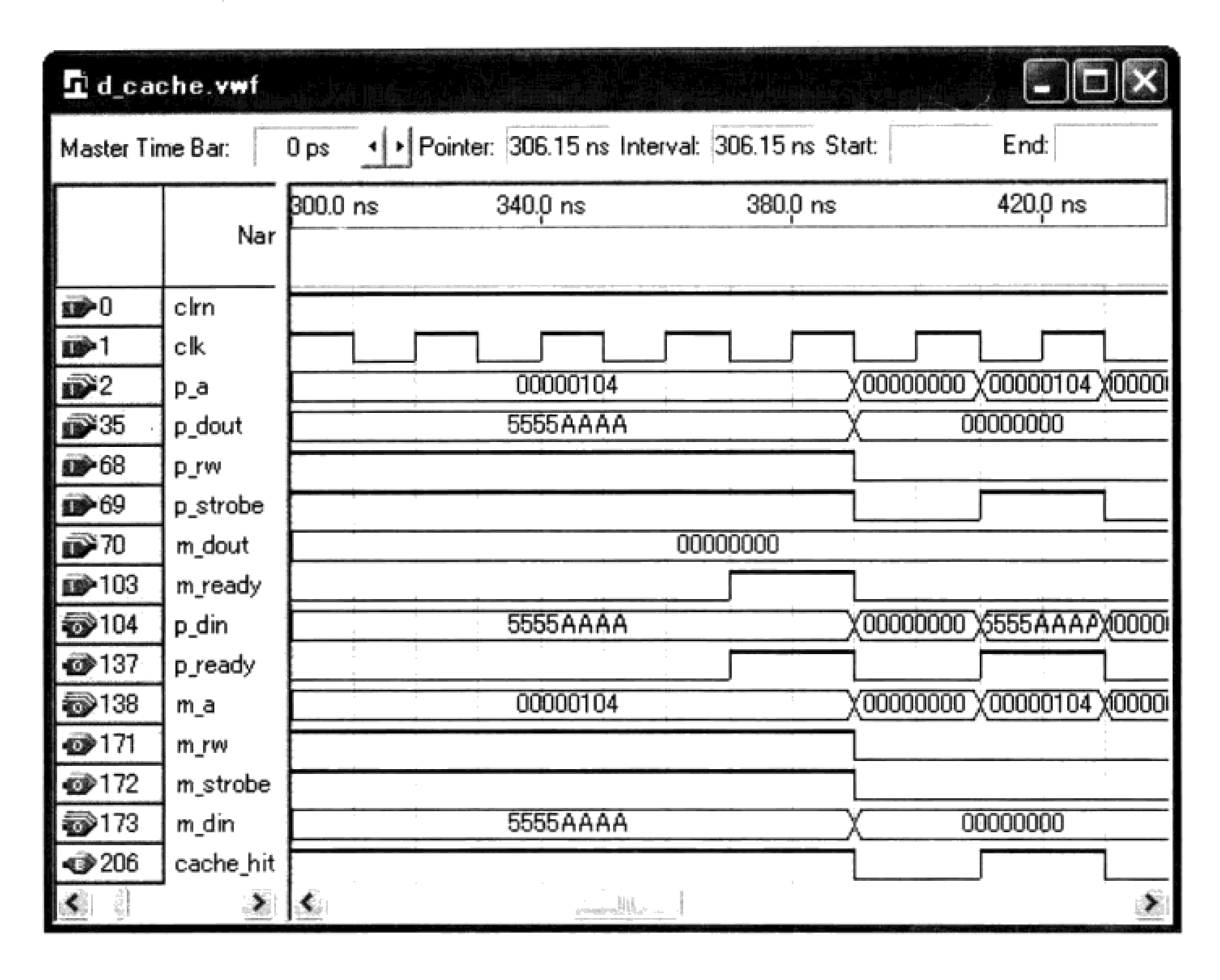


图 12.14 数据 Cache 仿真波形图 (写)

等，大致上它们的含义与虚拟地址相同。同样，存储器地址、物理地址、主存地址等又和实际地址有相同的含义。

12.3.1 虚拟存储器与主存的关系

我们知道，多个进程可以通过进程切换 (Context Switching) 的方式轮流运行，多线程 CPU 能同时运行多个进程。每个进程都使用从 0 开始的多至 4GB 的虚拟存储空间。图 12.15 示出的是两个进程的例子。由虚拟地址到实际地址的转换通过页表 (Page Table) 实现。页表的输入是进程号和虚拟页号，输出是主存的实际页号。图中主存的每页大小是 4KB，需要转换的是页号，页内偏移量不需转换。注意图中所示的是转换关系，实际上页表的输入只有一组信号，输出也只有一组信号。

把虚拟地址转换成实际地址这项工作由存储器管理部件 (Memory Management Unit，MMU) 完成。管理方法有分段管理 (Segmentation) 和分页管理 (Paging) 两种。图 12.15 示出的是分页管理。以下我们描述这两种管理方法。

12.3.2 分段管理

分段管理是把存储器分成若干段 (Segment)，为一个程序指定一个或几个“段寄存器” (Segment Registers)。这些寄存器指定当前段所在的主存的起始地址。每次访

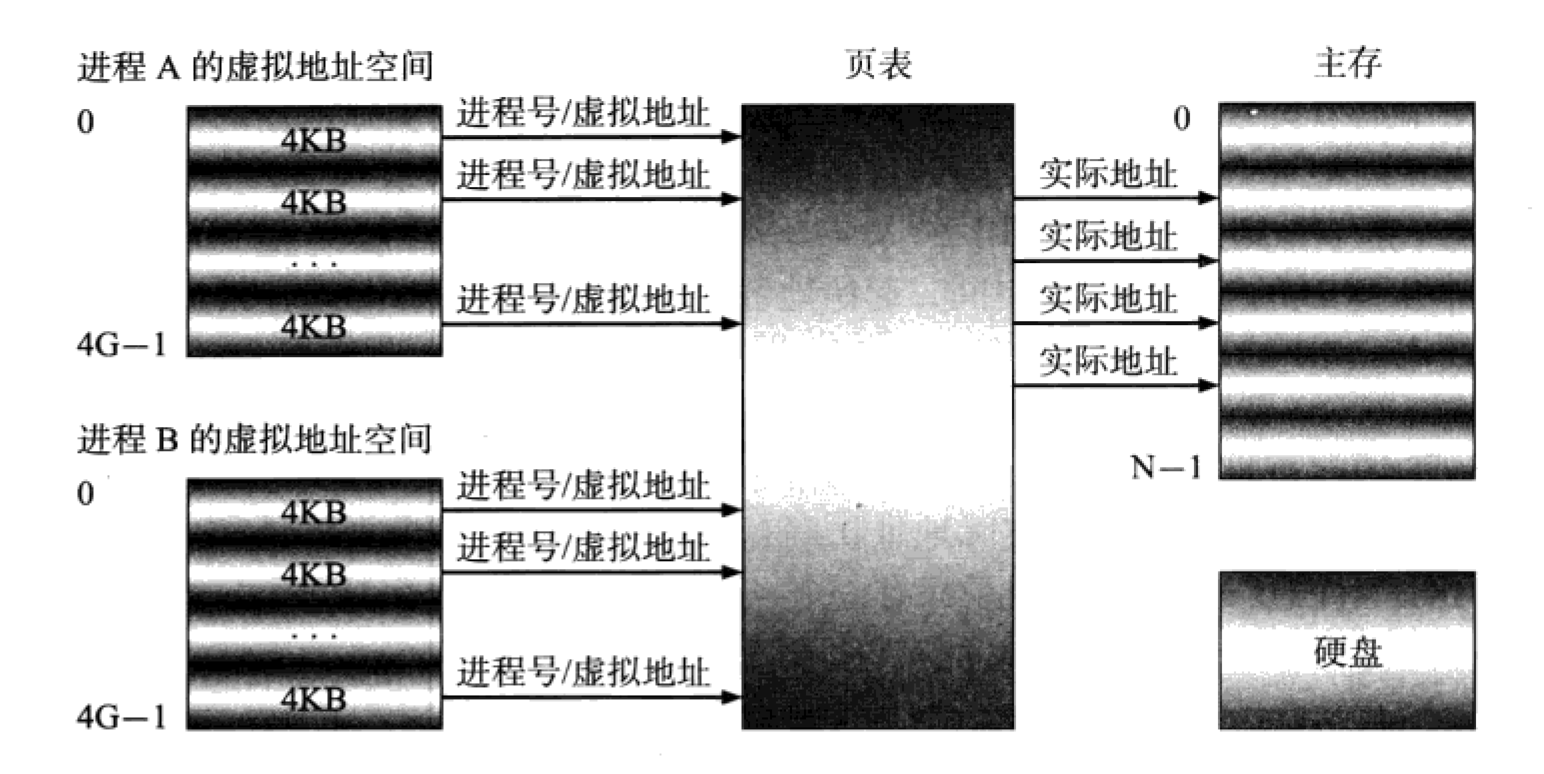


图 12.15　把虚拟地址转换成主存地址

问存储器时，主存的地址由段寄存器的内容与程序中的虚拟地址相加得到。比如 x86 就是这么做的。

除了主存的起始地址，段寄存器中可能还存有该段的大小以及对该段的访问控制信息。段的大小用于判断虚拟地址是否出界，而访问控制信息指出该段是否已在主存以及能否对其进行读写或执行等访问权限的信息。由此可见，分段管理中每段的大小是可以变化的。

12.3.3　分页管理

与分段管理不同，分页管理把存储器机械地分成若干页 (Page)，使用单一的虚拟地址，不需要段寄存器之类的东西。另外，虽然一页的大小是可以改变的，但是，一旦决定了，每页的大小就固定了，都是一样的。图 12.16 是分页管理的地址转换示意图，一页 4KB，页内地址有 12 位。

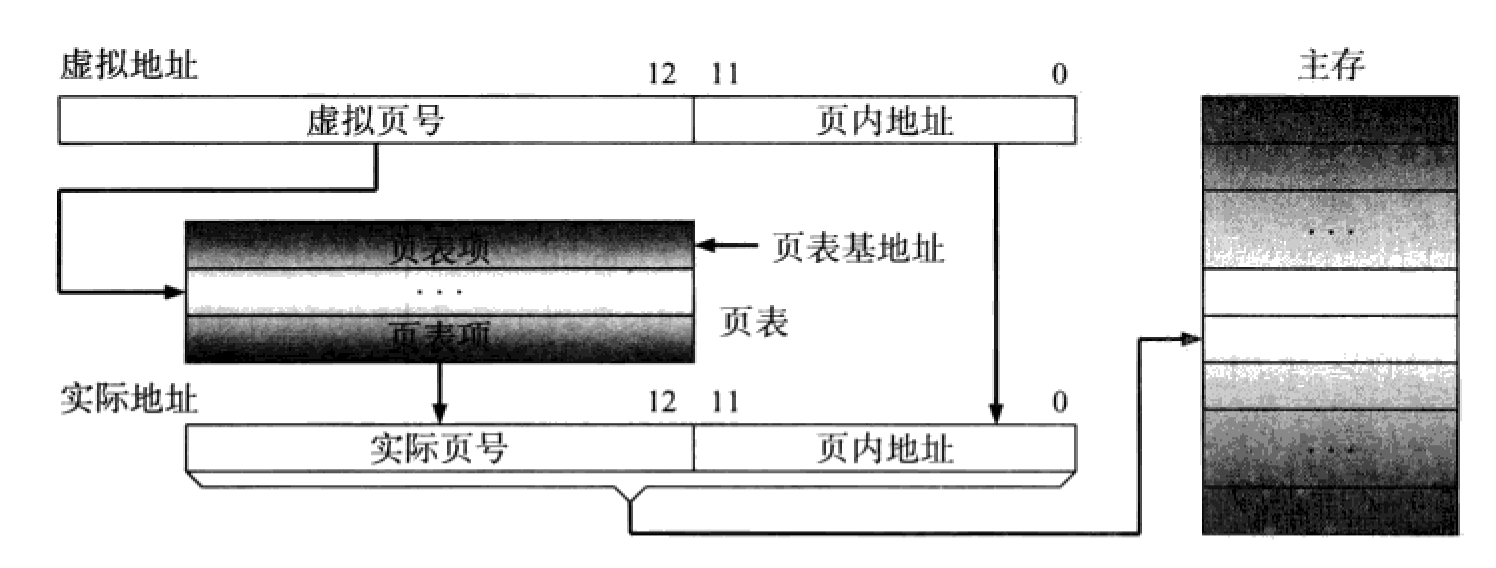


图 12.16　分页管理的地址转换

假设图 12.16 中的虚拟地址是 32 位。那么我们把高 20 位虚拟地址称做虚拟页号。使用虚拟页号作为地址访问页表 (Page Table)，从中得到实际地址的页号。实际页号与页内地址合起来就是实际地址。假设页表中的每一页表项 (Page Table Entry) 占用 4 个字节，则这个页表占用 $2^{20} \times 4 = 4\text{MB}$。问：页表在哪里？答：主存。页表需要占用 4MB 的连续的主存空间，起始地址由页表基地址 (实际地址) 指定。

使用 4MB 的连续的主存空间显然违背分页管理的精神。我们把 20 位的虚拟页号分成两部分：页目录地址 (10 位) 和页表地址 (10 位)，见图 12.17。我们首先使用页目录地址访问页目录，从中得到页表的起始地址，然后再使用页表地址访问页表，得到实际页号，最后再使用实际地址访问存储器。这样，一个页目录和一个页表都只占 4KB。注意，4KB 的页目录只有一个，但 4KB 的页表却有 1K 个。

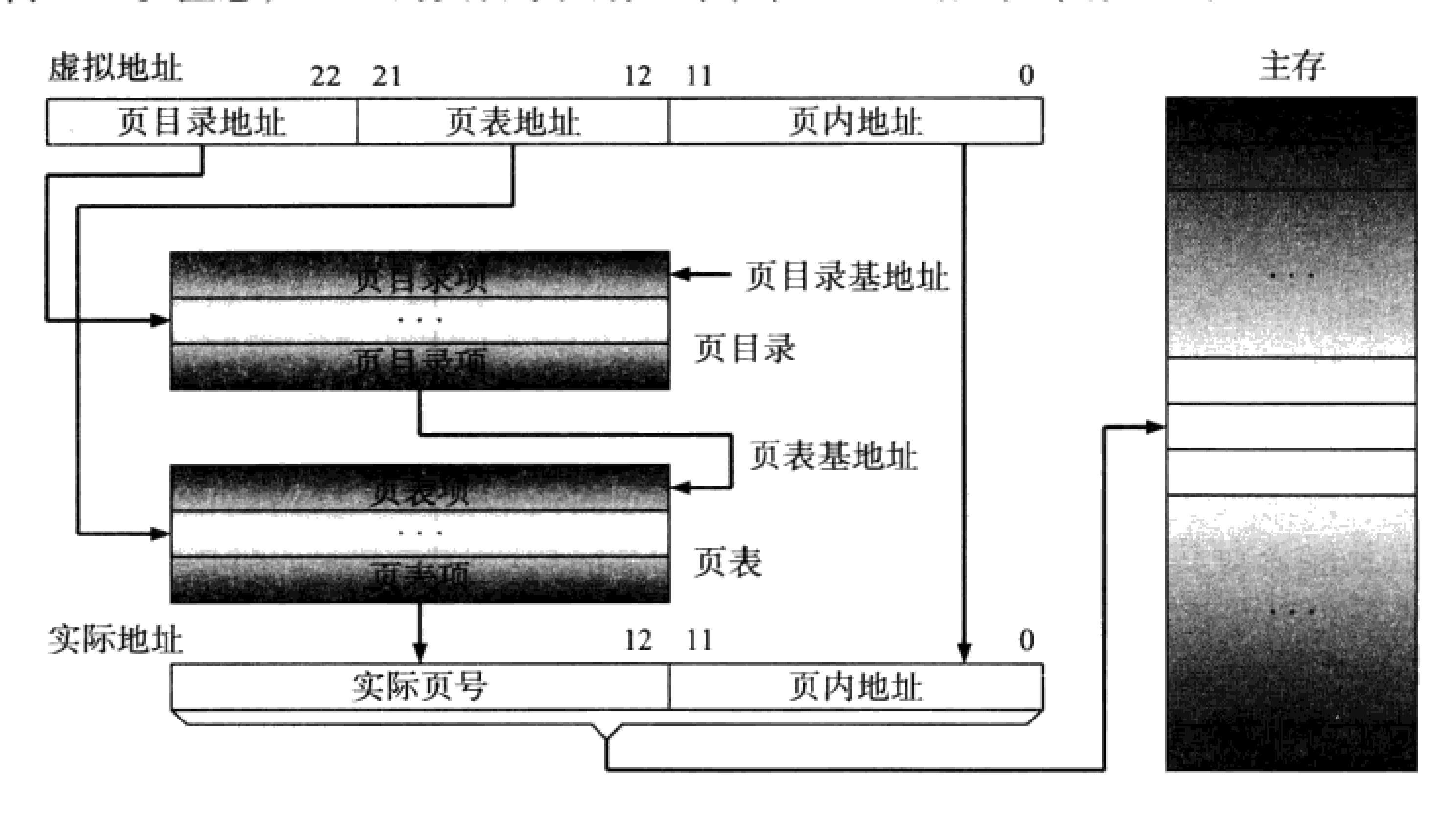


图 12.17 两级分页管理的地址转换

你看，为了访问一次数据存储器，先要访问一次页目录存储器，再要访问一次页表存储器，这时得到的才是实际地址。即，总共要访问三次主存。本来主存就比 CPU 的速度慢，乘以 3，就更慢了。有没有办法加快这个地址转换呢？有。

12.3.4 快速地址转换 TLB 及其电路设计

TLB (Translation Lookaside Buffer) 是一种与 Cache 极其相似的电路，只是它的目的是加快地址转换的速度。即，Cache RAM 中存放的不是数据而是存储器地址的实际页号部分。图 12.18 示出的是一种全相联映像的 TLB 的结构示意图。从 RAM 出来的实际页号与页内地址合起来是访问主存的实际地址。其他映像的 TLB 结构请参考相应的 Cache 结构。

图 12.19 示出的是 8 个 TLB 项的全相联 TLB 模块图。图中 CAM 是相联存储器，有 8 项，用虚拟页号来匹配 CAM 中的 vpn。若有一项匹配，vpn_found 输出 1，

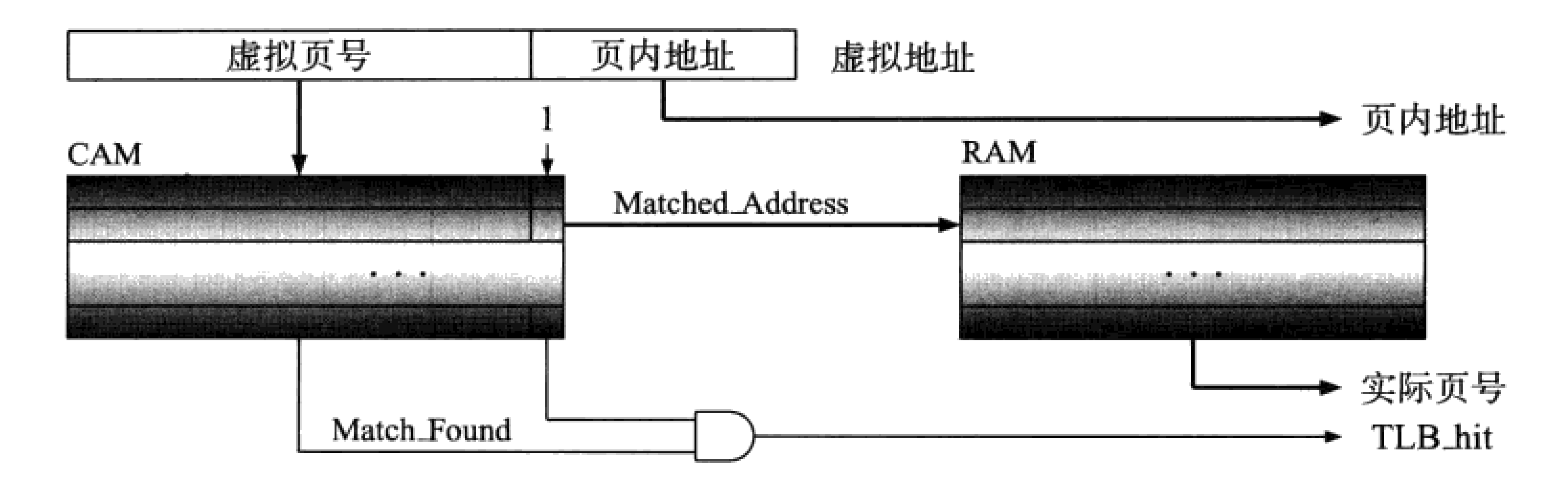


图 12.18 全相联映像的 TLB 示意图

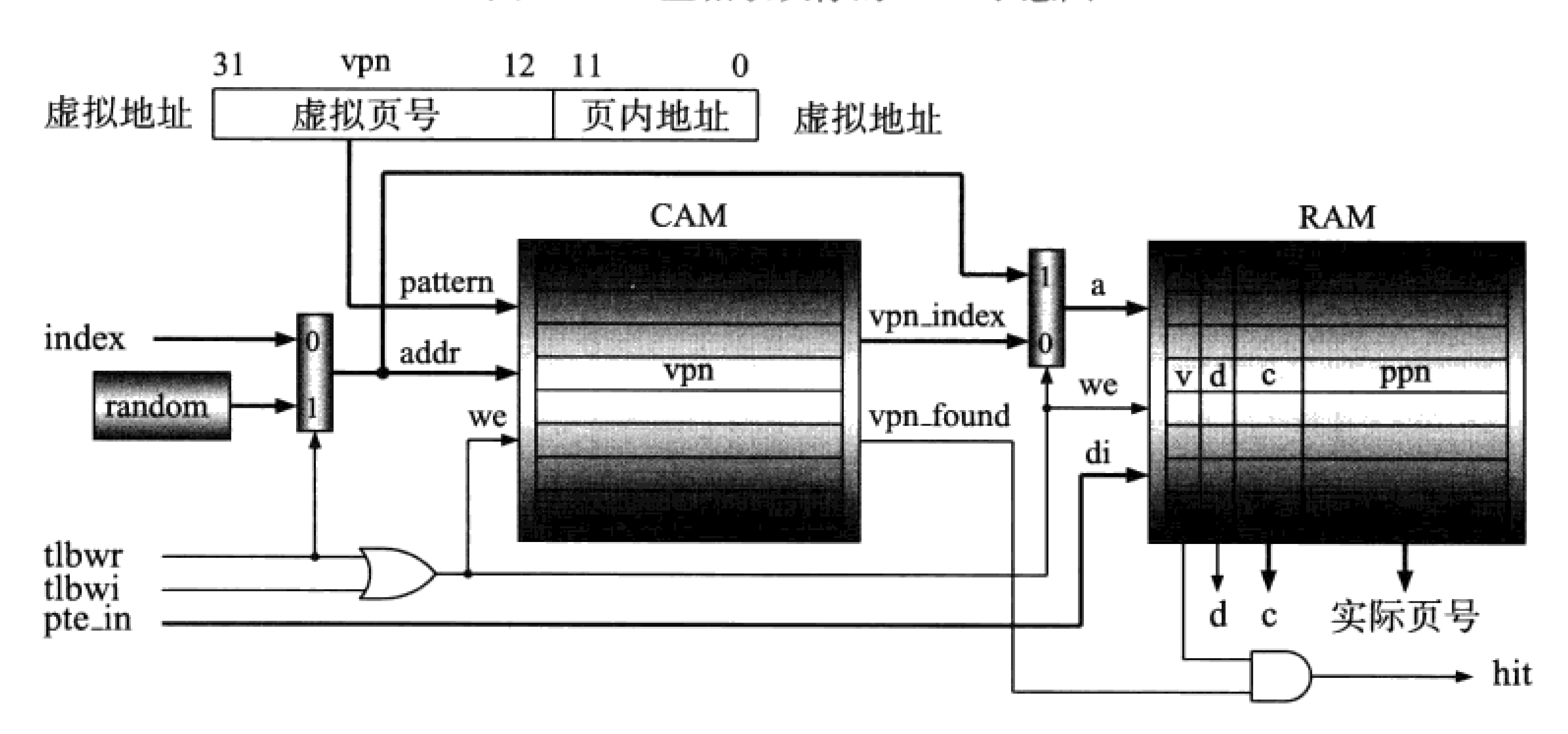


图 12.19 8 个 TLB 项的全相联 TLB 模块图

vpn_index 输出该项的地址，并作为访问 RAM 的地址，从 RAM 中得到实际页号。对 CAM 和 RAM 的写操作有两种方式：tlbwi 指令使用指定的地址 index；tlbwr 指令使用随机的地址 random。pte_in 是往 RAM 中写入的实际地址页号。hit 表示 TLB 命中。

图 12.20 是具体的电路实现。CAM 用 Altera 的 altcam 器件，RAM 用 lpm_ram_dq 器件。随机数使用一个 3 位的计数器。

以下是图 12.20 中的 ram8x24 模块的 Verilog HDL 代码。

```
module ram8x24 (address,data,inclock,outclock,we,q);
    input  [2:0]  address;
    input  [23:0] data;
    input  inclock;
    input  outclock;
    input  we;
    output [23:0] q;
    lpm_ram_dq ram_comp (.outclock (outclock),
```

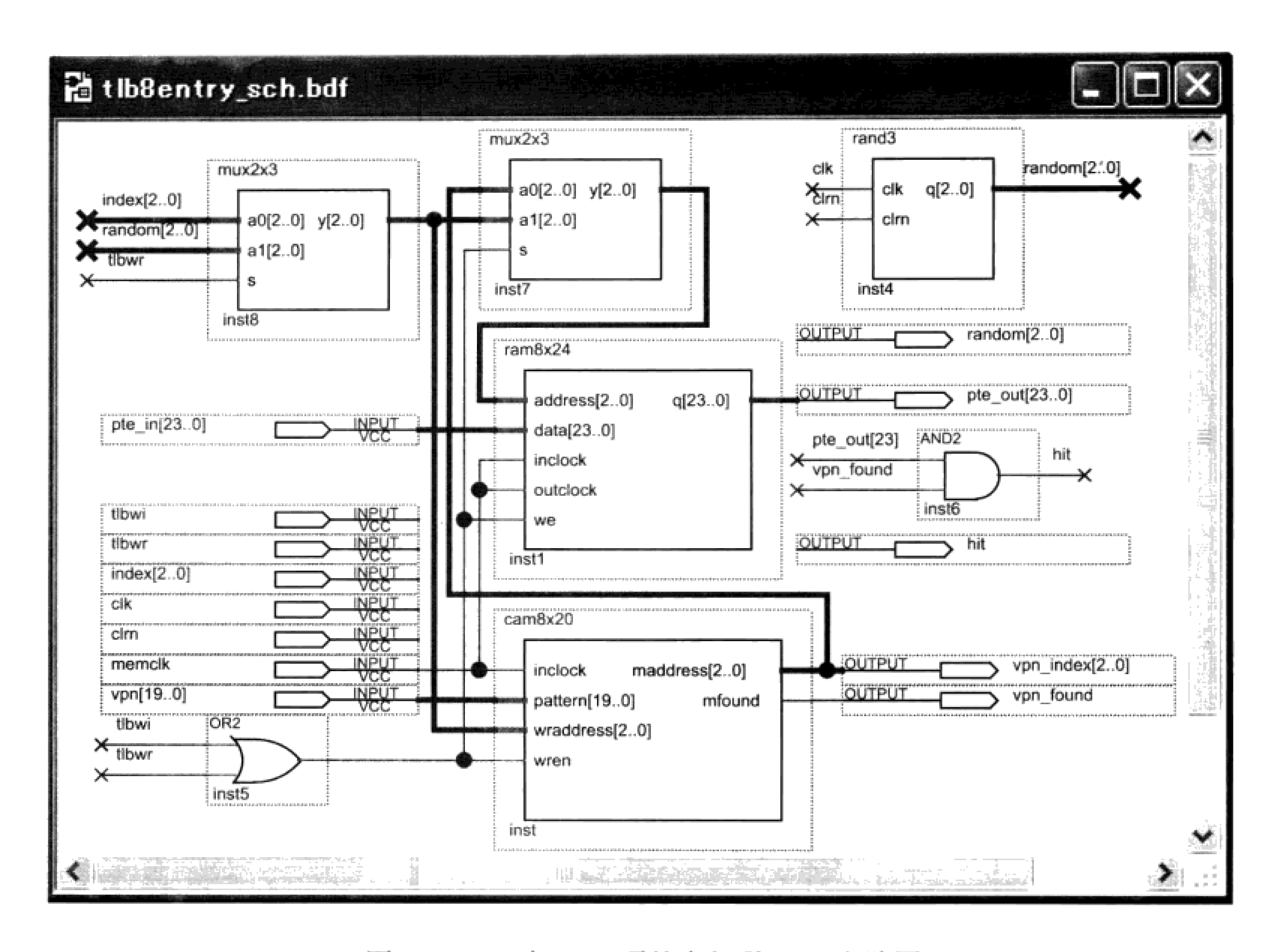


图 12.20　8 个 TLB 项的全相联 TLB 电路图

```
                        .address (address),
                        .inclock (inclock),
                        .data (data),
                        .we (we),
                        .q (q));
    defparam    ram_comp.lpm_width    = 24,
                ram_comp.lpm_widthad = 3,
                ram_comp.lpm_indata  = "registered",
                ram_comp.lpm_outdata = "registered",
                ram_comp.lpm_type     = "lpm_ram_dq",
                ram_comp.lpm_address_control = "registered";
endmodule
```

以下是图 12.20 中的 cam8x20 模块的 Verilog HDL 代码。

```
module cam8x20 (inclock,pattern,wraddress,wren,maddress,mfound);
    input  inclock,wren;
    input  [19:0] pattern;
    input  [2:0]  wraddress;
    output [2:0]  maddress;
    output mfound;
```

```
    altcam cam (.wren (wren),
                .inclock (inclock),
                .pattern (pattern),
                .wraddress (wraddress),
                .maddress (maddress),
                .mfound (mfound));
    defparam cam.lpm_type        = "altcam",
             cam.match_mode      = "single",
             cam.numwords        = 8,
             cam.output_aclr     = "off",
             cam.output_reg      = "inclock",
             cam.pattern_aclr    = "off",
             cam.pattern_reg     = "inclock",
             cam.width           = 20,
             cam.widthad         = 3,
             cam.wraddress_aclr  = "off",
             cam.wrcontrol_aclr  = "off";
endmodule
```

图 12.21 是使用 tlbwi 写入 TLB 时的仿真波形，图 12.22 是命中时的仿真波形及没命中时使用 tlbwr 写入 TLB 时的仿真波形。

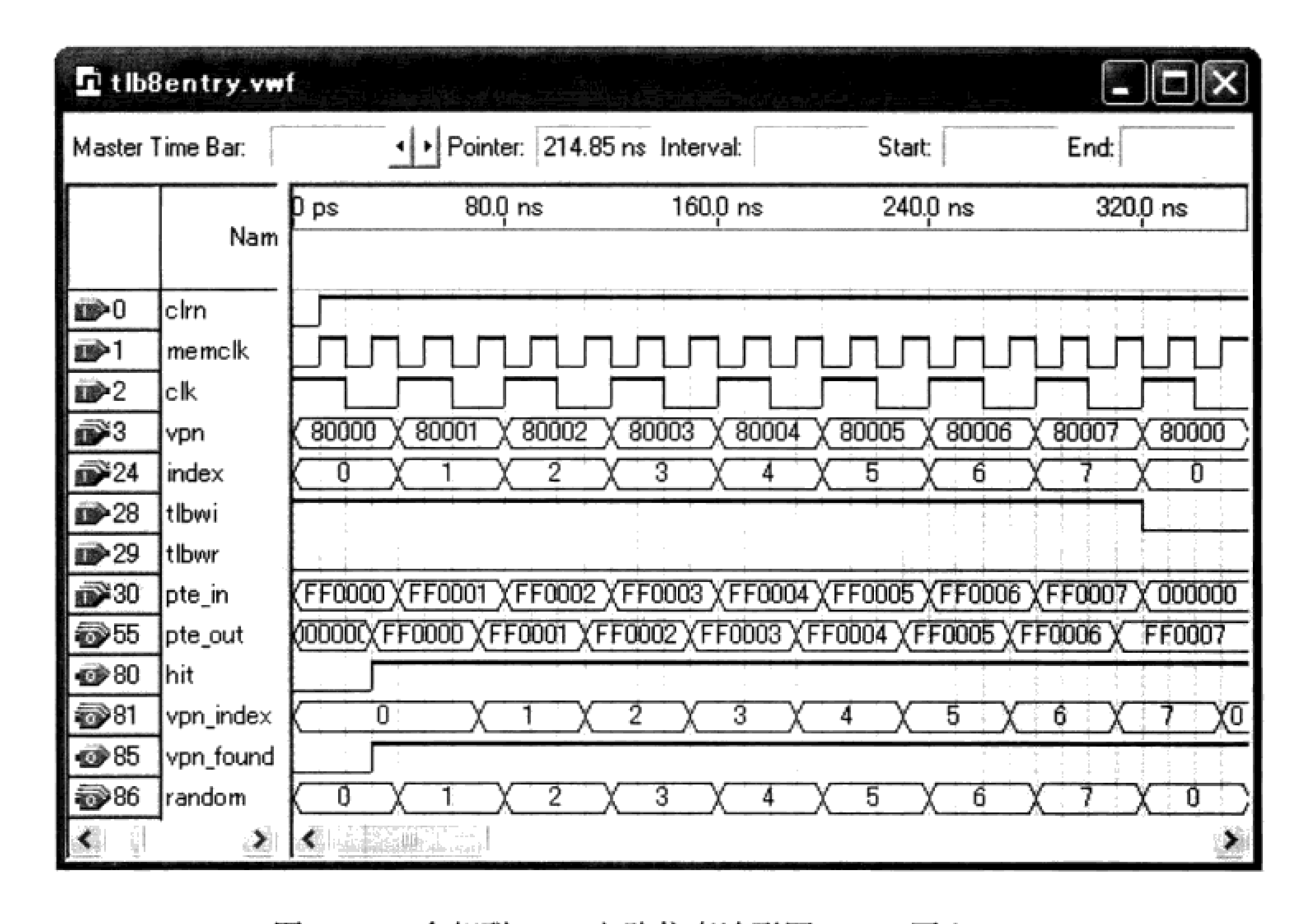


图 12.21　全相联 TLB 电路仿真波形图 (tlbwi 写入 TLB)

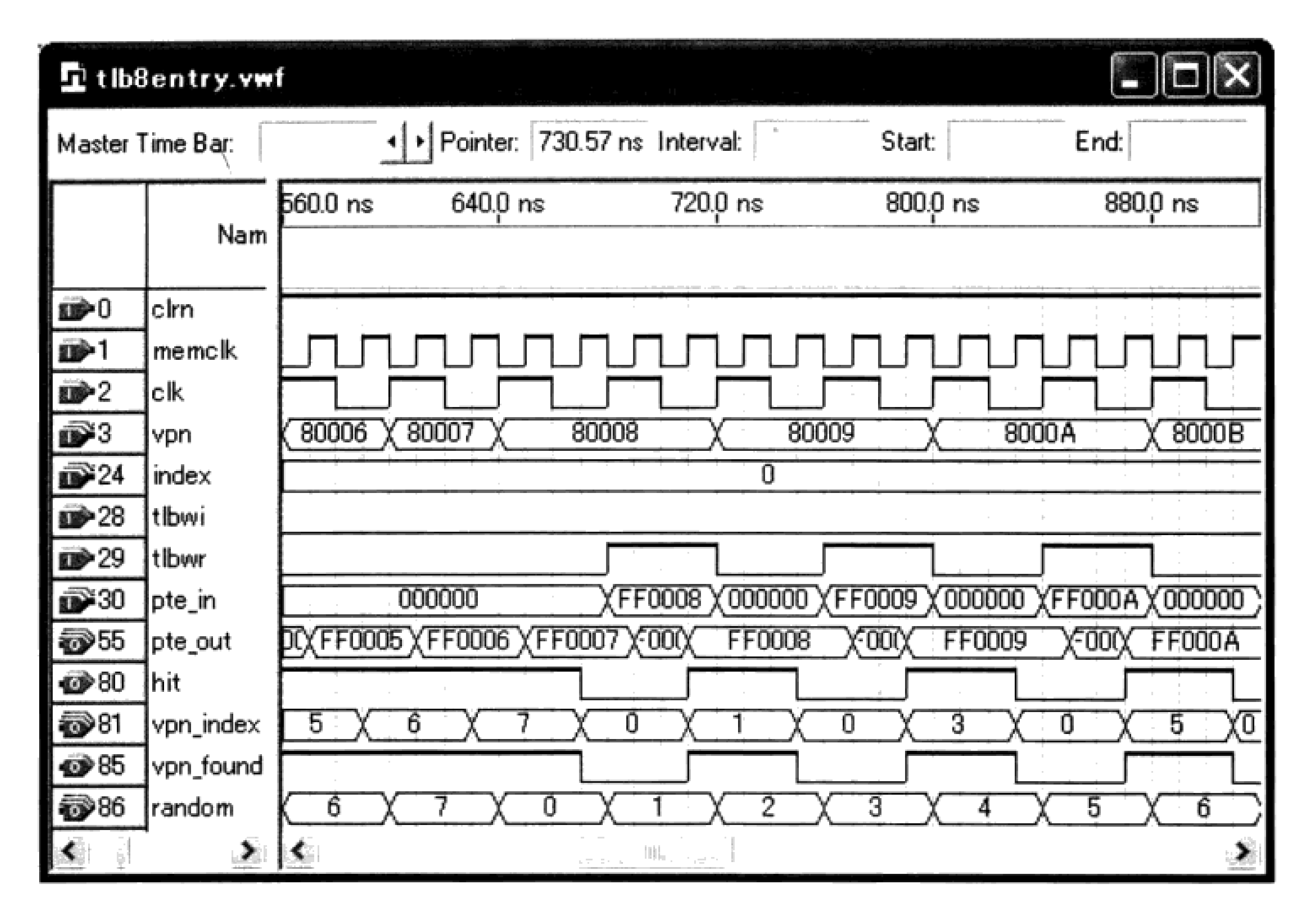


图 12.22 全相联 TLB 电路仿真波形图 (tlbwr 写入 TLB)

12.3.5 TLB 与 Cache 的并行访问

假设我们使用 k 路组相联映像的 Cache 且 Cache 的标志是实际地址的高位部分。在分页管理的情况下，假设页的大小为 2^m 字节。当 Cache 的容量不大于 $2^m \times k$ 字节时，访问 Cache 和访问 TLB 可以同时进行，见图 12.23。

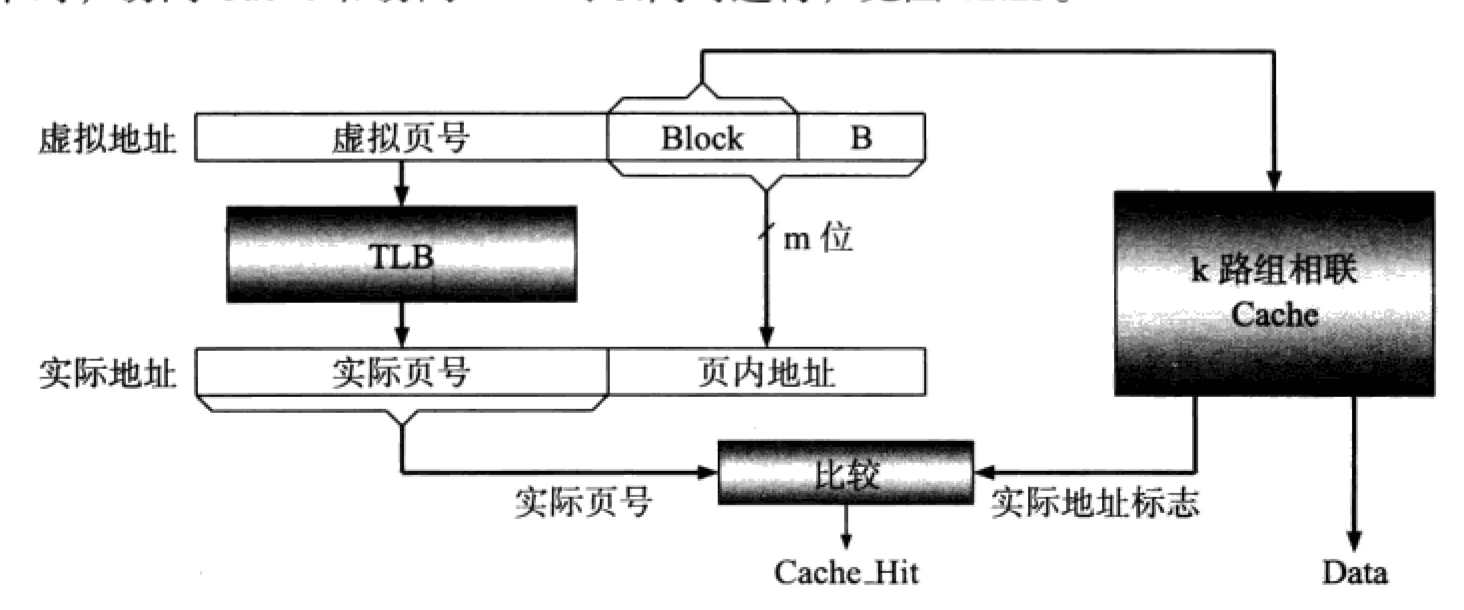


图 12.23 TLB 与 Cache 可以并行访问的条件

同时访问的意思是：使用虚拟页号查找 TLB 得到实际页号，使用不需转换的页内地址访问 Cache。假设二者的访问时间相同，则实际页号和 Cache 标志同时输出，

然后比较它们以确定 Cache 是否命中。例如，m = 12、k = 4 时，我们可以使用多至 16KB 的 Cache。如果 Cache 的容量超出这个界限，访问 Cache 时不得不使用 TLB 输出的实际页号的低若干位了，从而延长了 Cache 的访问时间。

以上是使用实际地址访问 Cache 的情况。那么访问 Cache 直接使用虚拟地址 (地址标志也是虚拟地址的高位部分) 是否可行呢？如果可行，它又会有怎样的结构呢？会出现什么问题吗？请读者思考并给出设计方案。

12.4 MIPS 基于 TLB 的虚拟地址转换机制

MIPS 使用 TLB 把虚拟地址转换成实际地址。特点是，MIPS 提供了用于 TLB 维护的指令和寄存器。

12.4.1 MIPS 的虚拟地址空间

MIPS 体系结构把 CPU 的运行状态分成三种：用户方式 (User Mode)、管理方式 (Supervisor Mode) 和核心方式 (Kernel Mode)。为了讨论方便，我们在这里忽略管理方式。图 12.24 是 MIPS 的虚拟地址空间与实际地址空间的对应关系。图中把 4GB 的虚拟地址空间分成了 4 段 (Segments)：kuseg (2GB)、kseg0 (512MB)、kseg1 (512MB) 和 kseg2 (1GB)。核心方式可以使用所有的段，而用户方式只能使用 kuseg。虽然称做段，但 MIPS 的存储器管理用分页管理方式。

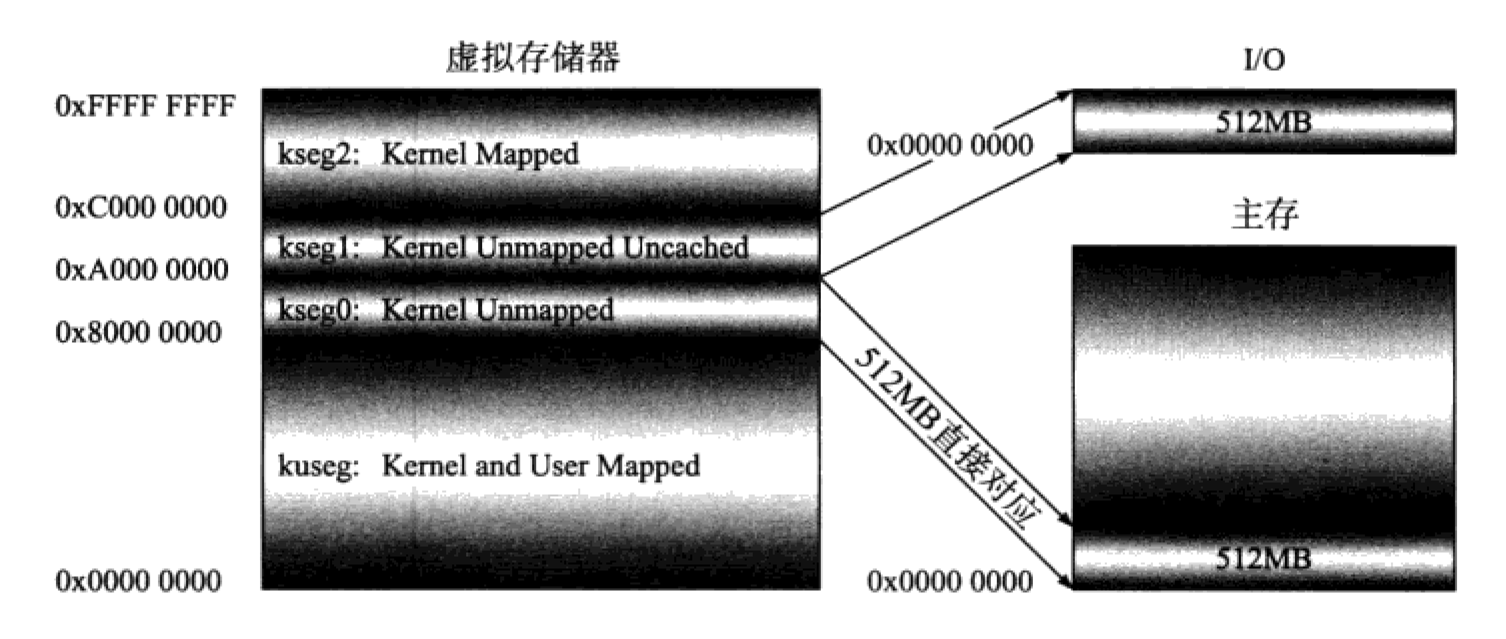


图 12.24 MIPS 虚拟地址空间映像

图中的“Mapped”表示虚拟地址要经过 TLB 转换 (kseg2 和 kuseg)、“Unmapped”不允许使用 TLB (kseg1 和 kseg0)、“Uncached”表示不能往 Cache 里放 (kseg1)。kseg0 (0x8000 0000 ~ 0x9FFF FFFF) 虽然不经过 TLB，但也不是直接使用，而是直接映像到主存的 0x0000 0000 ~ 0x1FFF FFFF。这相当于某些其他 CPU 的“Real Mode”。如果你有机会阅读为 MIPS 准备的操作系统的源程序，你会发现很多代码的起始地址是 0x8xxx xxx0 (虚拟地址)，把最高位的 1 改成 0 就是实际的主存地址。

注意，kseg1 也直接映像到实际地址空间的 (0x0000 0000 ～ 0x1FFF FFFF)。这岂不是要引发战争吗？不会，这段地址用于访问 I/O。这也是为什么它不被允许使用 Cache 的原因。你往存储器的某个单元写入一个数据，然后再读它：数据还是那个数据。但是，I/O 地址空间不具有这个特性：你往一个 I/O 地址写的可能是“控制”信息，而从相同的地址读来的可能是“状态”信息。所以禁止 kseg1 使用 Cache。MIPS 读写 I/O 也是使用诸如 lw 和 sw 之类的存储器访问指令。这就是所谓的“存储器映像的 I/O”。这一点与 x86 不同， x86 有专门的 I/O 指令。

你看，当机器刚启动时，TLB 没被初始化也没关系：操作系统既有从最低地址开始的 512MB 的主存可以使用，又有 512MB 的 I/O 地址可用。作为一个系统软件工程师，没什么好抱怨的了吧。接下来的工作是初始化页表以及 TLB，好让 kuseg 和 kseg2 也能有存储器可用。

12.4.2 MIPS TLB 的构成

MIPS 所有的进程都有相同的虚拟地址空间。为了能够区分开它们，MIPS 为每个进程指定一个不同的“地址空间标示符”ASID (Address Space Identifier)。ASID 有 8 位，可以被看作是“扩展的虚拟地址”的高位部分。这样，不同的进程就有不同的“扩展的虚拟地址”空间了。在用 TLB 做地址转换时，ASID 也参加比较。但是，在某些情况下，操作系统又希望所有的进程“共享”相同的虚拟地址空间。为此，TLB 中增加了一位“全局”标志 G (Global)。如果 G 被设置为 1，则在转换时忽略 ASID。

使用 TLB 的目的是为了加快从虚拟地址到实际地址的转换。MIPS TLB 使用全相联映像结构，它的每一项由“虚”和“实”两部分组成，见图 12.25。

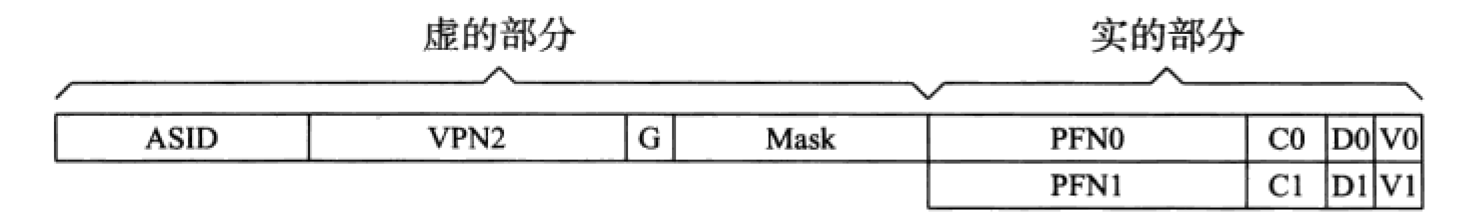


图 12.25 一个 TLB 项的内容

虚的部分包含有 ASID、全局标志 G、虚拟页号 (Virtual Page Number) VPN2 (实际的意义是 VPN / 2，因为每一项对应两个实际的存储器页：偶页和奇页) 和用来指定一页大小的页屏蔽域 Mask。实的部分包含有偶页和奇页两组实际地址页号 PFN (Physical Page Frame Number) 及相关的控制和状态信息 (C、D 和 V)。偶页组编号为 0、奇页组编号为 1。V (Valid) 表示该实际地址页号是否有效；D (Dirty) 指出该页存储器的内容是否被修改过，比如执行 sw 指令；C (Cache Coherency) 是 Cache 一致性信息 (见表 12.1)。

CP0 寄存器 PageMask、EntryHi、EntryLo0 和 EntryLo1 中定义的每个域与 TLB 项中的每个域具有相同名称。这些寄存器被用来访问 TLB 项。TLB 偶页项来自于 EntryLo0 寄存器，奇页项来自于 EntryLo1 寄存器。有一处稍微不同：TLB 项中的 G

表 12.1 Cache 页一致性属性

C	Cache 页一致性属性
0～1	没有定义、具体实现时可由厂家自定义
2	该页存储器的内容不准往 Cache 里面放
3	该页存储器的内容可以往 Cache 里面放 (一般的存储器)
4～7	没有定义、具体实现时可由厂家自定义

是 EntryLo0 和 EntryLo1 中的两个 G 的逻辑“与”。

在表 12.1 中没有定义的 C 的值可以由 MIPS CPU 的生产厂家自己来定义，比如 IDT79RC32355 CPU[4] 中对 C 域有如表 12.2 所示的定义。IDT79RC32355 手册中没有讲采用写回策略时 (C = 3) 对写不命中如何处理，但作者猜测是把相应的数据块取到 Cache 中 (Write Allocate)，以便下次访问到它时命中，反正它采用的是一次性写回的策略，不用每次执行 sw 指令时都要写入存储器 (Cache 命中时只写 Cache)。

表 12.2 IDT79RC32355 Cache 页一致性属性

C	Cache 页一致性属性
0	可以往 Cache 里面放、写透、写不读入 (Write Through, No Write Allocate)
1	可以往 Cache 里面放、写透、写前读入 (Write Through, Write Allocate)
2	不准往 Cache 里面放
3	可以往 Cache 里面放、写回 (Write/Copy Back)
4～7	保留

下面讲 TLB 项中的 Mask 域到底是干什么用的。Mask 对应于 CP0 第 5 号寄存器 PageMask (见图 12.26) 中的 Mask 域。它有 16 位，占用 PageMask 寄存器的 [28:13] 位。如果这些位全部为 0，则存储器页的大小为 4KB。即，实际地址的低 12 位 ([11:0]) 与虚拟地址的低 12 位相同。虚拟页号 (虚拟地址中的 [31:12] 位) 需要转换。但由于一个 TLB 项有奇偶两组实际页号，由虚拟地址的 [12] 位来选择，所以参与转换的虚拟地址位就变成了 [31:13]。MIPS CPU 可以通过设置适当的 Mask 值来改变存储器页的大小。当 Mask 位为 1 时，相应位置的虚拟地址就不需转换了，即一页存储器变大了。不像第 6 章中的中断屏蔽 IM，这里的 Mask 是真正意义上的“屏蔽”。

31 29	28 13	12 0
0	Mask	0

图 12.26 PageMask 寄存器 (CP0 寄存器 5) 的格式

表 12.3 列出了 Mask 的取值和与其相对应的存储器页的大小。最大的页就是 256MB 了。把 4GB 定为存储器页的大小不能说是愚蠢的，它意味着不需要做地址转换，比如在大部分嵌入式的应用中就不转换地址，而且存储器也不需要那么大。表中最右列的 S 用于选择实际地址页号 0 还是页号 1，它只有一位，括号中的数字是它

表 12.3 PageMask 寄存器中的 Mask 取值

页的大小	位 28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	S
4KB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	[12]
16KB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	[14]
64KB	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	[16]
256KB	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	[18]
1MB	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	[20]
4MB	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	[22]
16MB	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	[24]
64MB	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	[26]
256MB	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	[28]

在虚拟地址中所处的位号。在具体的 MIPS CPU 的设计中，虽然要做地址转换，但你也可以不使用 PageMask 寄存器。这时存储器页的大小就是固定的 4KB 了。

MIPS CP0 寄存器 EntryLo0、EntryLo1 和 EntryHi 的格式如图 12.27 和图 12.28 所示。EntryLo0 和 EntryLo1 的格式完全相同，主要内容是存储器地址的实际页号；EntryHi 的主要内容是虚拟页号。

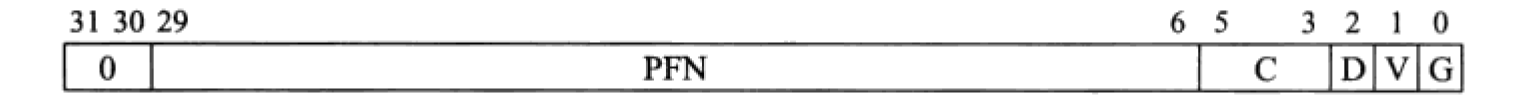


图 12.27 EntryLo0 和 EntryLo1 寄存器 (CP0 寄存器 2 和 3) 的格式

31 13 12 8 7 0

VPN2 | 0 | ASID

图 12.28 EntryHi 寄存器 (CP0 寄存器 10) 的格式

两个图中所有的域的意义已经在对 TLB 项的描述中简单地提过了，这里再多讲几句。PFN (Physical Page Frame Number) 是存储器实际地址的页号，有 24 位。假设每页为 $4KB = 2^{12}B$，PFN 和页内偏移量两部分地址拼凑起来就是存储器的实际地址，有 $24 + 12 = 36$ 位，能访问 $2^{36} = 64GB$ 的存储器。你的机器有那么多的内存吗？没有吧。怎么办？简单！把地址的高位扔了。实际上，操作系统能侦察出你的机器的家底有多厚。没那么多存储器的话，操作系统也不会把进程的虚拟地址转换到你力所不能及的界外。

TLB 项中的 PFN 从哪里来？是从那寄存器 EntryLo0 或 EntryLo1 中来。寄存器 EntryLo0 或 EntryLo1 中的 PFN 又从哪里来？对面的 IU 写过来。写什么呢？IU 从哪里能得到它呢？答案是从我们已经讲过的用于分页存储器管理的动态页表中得到。

图 12.27 中有一位 V (Valid)，为 1 时表示可以使用该项做地址转换；为 0 时产生

TLBL 或 TLBS 异常 (见第 6 章表 6.1 对 Cause 寄存器中 ExcCode 的定义)。

还有就是图 12.27 中的 D 又是负的什么责呢？D：Dirty，有不干不净的意思。一页存储器不干净是什么意思呢？存储器本来是干净的 (D = 0)，当 CPU 执行诸如 sw 类的存数据指令第一次往该页写数据时，存储器就不干净了。这是一件再普通不过的事情了，但还是要向 CPU 报警：产生 Mod 异常 (也见表 6.1)。如果该页存储器已经不干净了 (D = 1)，就可以随便写了，再也不会报警了。

操作系统可以使用它来实现一些存储器分页管理的算法，比如页替换算法等。当不干净的页被替换掉时，要把它保存到硬盘的 Swap 区域 (文件)。这一点与 Cache 的管理非常类似：存储器管理使用全相联映像方式，并且使用写回策略。其他 CPU 的 TLB 项中经常会有一位 Reference 或 Used 位，用于实现类似于 LRU 的存储器页替换算法。但 MIPS 没有 Reference 位，似乎只能使用随机替换策略了。

12.4.3 MIPS 虚拟地址转换

当对一个虚拟地址进行转换时，把虚拟页号 VPN2 以及当前进程的 ASID 同时与 TLB 中的所有项进行比较 (全相联)。如果以下条件全部满足时，TLB 命中，从而得到实际地址页号。

1) 由虚拟地址中的 S 位选中的有效位 V0 或 V1 的值是 1 (有效)，S 位在虚拟地址中的位置由 PageMask 寄存器中的 Mask (也在 TLB 中) 决定，见表 12.3；
2) 当前进程的 ASID 与 TLB 项中的 ASID 匹配，或者 TLB 项中的 G 为 1；
3) 去掉被屏蔽的位，虚拟页号 VPN2 与 TLB 项中的 VPN2 相同。屏蔽哪些位由 PageMask 寄存器中的 Mask 域指定。MIPS 利用 PageMask 寄存器可以改变存储器页的大小。如果 CPU 中没有设计 PageMask 寄存器，则认为 Mask 为 0，虚拟页号 VPN2 的任何一位都不被屏蔽。这时存储器页的大小为 4KB。

见图 12.29，如果 TLB 命中，未经转换的页内地址与从 TLB 得到的实际地址页号合在一起，形成访问存储器的实际地址。为了清楚地表示出虚拟地址中的 S 位，屏蔽位 Mask 画在了虚拟地址的下面，但要注意它是在 TLB 中。

12.4.4 MIPS TLB 维护指令

MIPS CPU 不能对 TLB 项直接访问。见图 12.30，CPU 要想读写 TLB 项，必须使用 CP0 中的若干寄存器和一些特殊的指令 (tlbp、tlbr、tlbwi 和 tlbwr)。

1. 与 TLB 维护指令有关的另外三个寄存器

我们已经介绍过了几个与 TLB 有关的 CP0 寄存器，比如 PageMask、EntryLo0、EntryLo1 和 EntryHi 寄存器。以下再介绍另外三个与 TLB 维护指令有关的寄存器，它们是 Index 寄存器 (CP0 寄存器 0)、Random 寄存器 (CP0 寄存器 1) 和 Wired 寄存器 (CP0 寄存器 6)。图 12.31 示出的是这三个寄存器的格式。

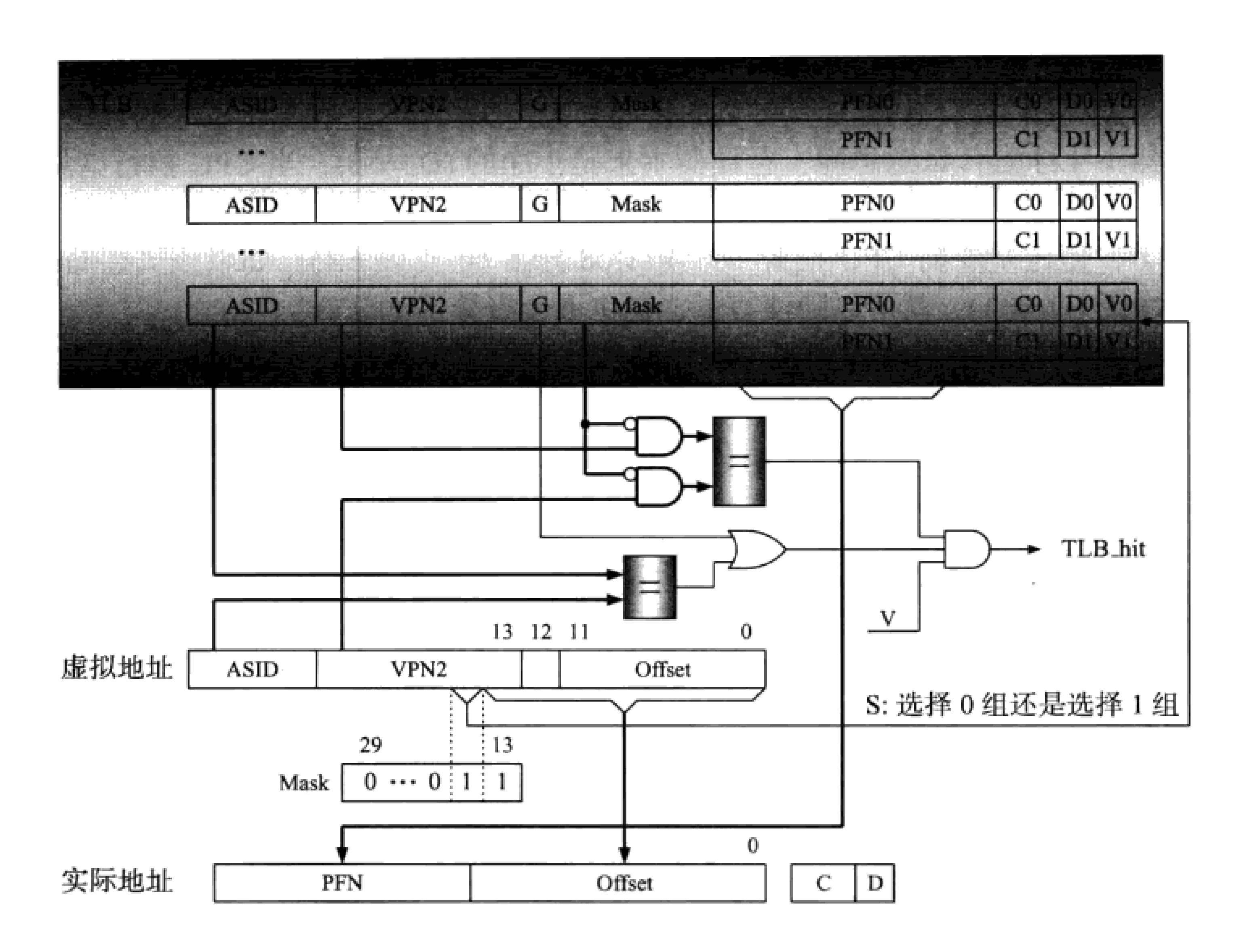


图 12.29　基于 TLB 的虚拟地址转换 (与所有的 TLB 项同时比较)

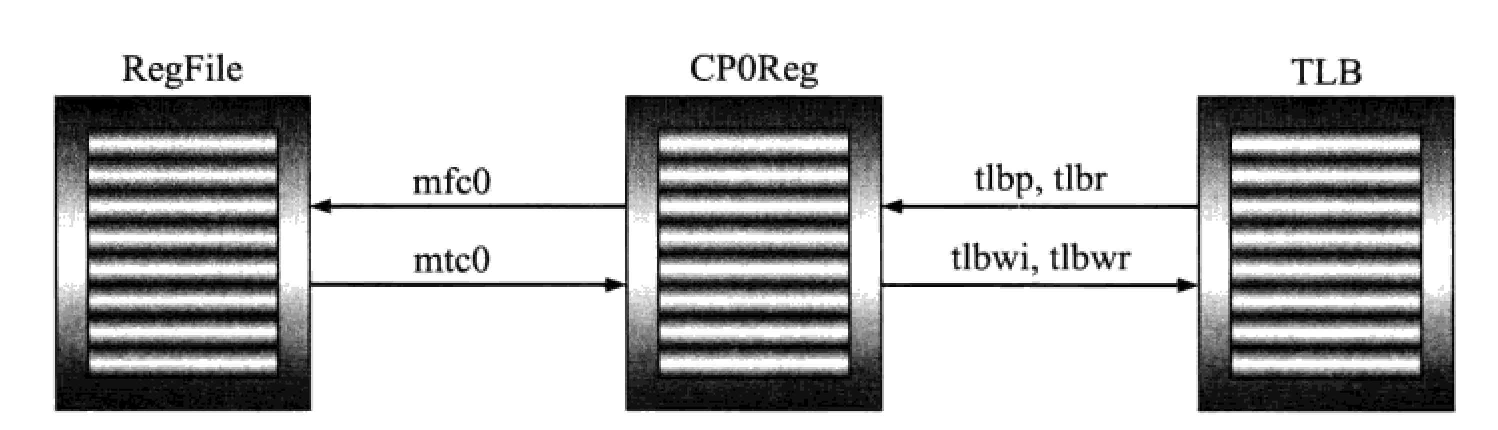


图 12.30　CPU 访问 TLB 必须经过 CP0 寄存器

	31	30 … n	n−1 … 0
Index:	P	0	Index
Random:	0		Random
Wired:	0		Wired

图 12.31　Index、Random 和 Wired 寄存器的格式

TLB 中有 $N = 2^n$ 个 TLB 项，每项都有一个 n 位的编号，相当于地址。Index

域就是被用来“指出”或“指定”这个地址的。比如 CPU 想修改某一 TLB 项中的内容，CPU 先要执行 mtc0 指令把那一项的地址写入 Index 寄存器，然后执行 tlbwi 指令。这是“指定”的意思。另外，CPU 有时也想调查一下 TLB 中有没有一项与 EntryHi 寄存器的内容匹配。如果有，硬件把那一项的地址写入 Index 域并把 P 位清零。这是“指出”的意思。CPU 可以使用 mfc0 指令把 Index 寄存器的内容读过来。

Random 寄存器是一个只能由 CPU 读但不能写的寄存器。我们知道 TLB 项的内容可能要被新的内容替换掉。替换掉哪一项呢？MIPS CPU 支持随机替换策略，即由硬件产生一个随机数，把这个随机数存放在 Random 寄存器。这个随机数的功效与 Index 相同，也是指定 TLB 项的地址。CPU 可以使用一条 tlbwr 指令来修改由 Random 寄存器指定的 TLB 项。

如果某些 TLB 项属于“重点保护单位”，不能被“随机”地替换掉，怎么办呢？答案是使用 Wired 寄存器。图 12.32 给出 Wired 寄存器的意义。假设 Wired 寄存器的内容是 i，则 TLB 的 0 ～ i − 1 项是重点保护单位。

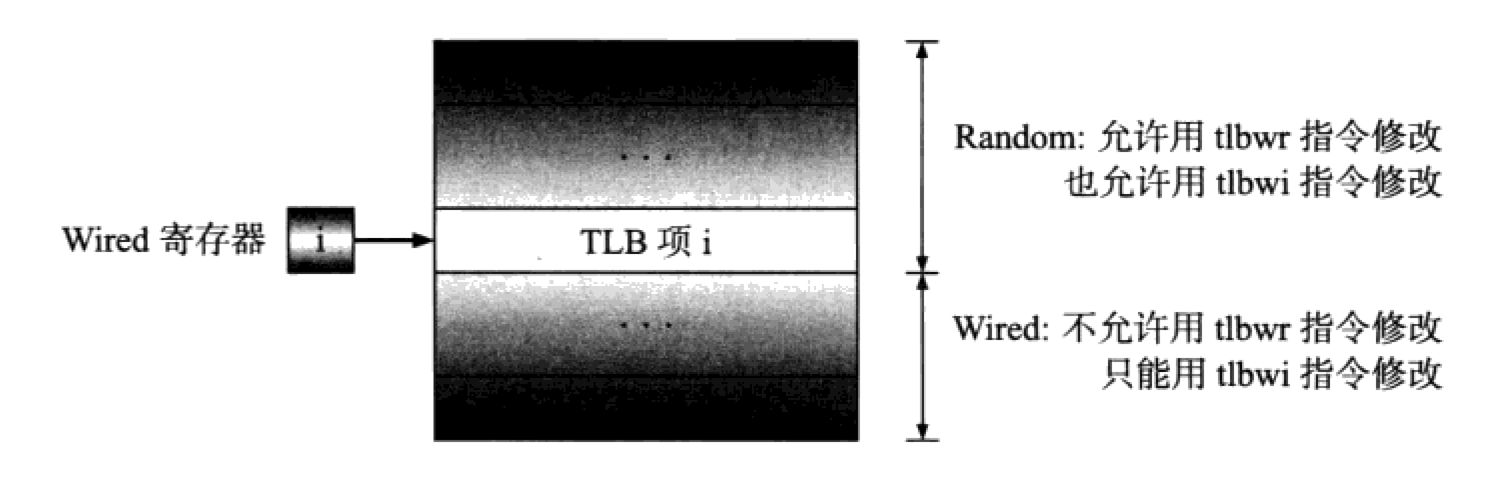


图 12.32 Wired 的意义

MIPS 用于 TLB 维护的指令有 4 条。使用 Index 寄存器修改 TLB 项的指令是 tlbwi；使用 Random 寄存器修改 TLB 项的指令是 tlbwr。还有两条指令是 tlbp 和 tlbr。它们的指令格式见图 12.33。

指令	31–26	25	24–6	5–0
tlbp	0 1 0 0 0 0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 1 0 0 0
tlbr	0 1 0 0 0 0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 1
tlbwi	0 1 0 0 0 0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1 0
tlbwr	0 1 0 0 0 0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 1 0

图 12.33 MIPS 4 条用于 TLB 管理的指令

2. tlbp (Probe TLB for Matching Entry) 指令

tlbp 指令检查是否有一 TLB 项与 EntryHi 寄存器的内容匹配 (相同)。如果有，则把该 TLB 项的号码 (地址) 存入 Index 寄存器，最高位 P (Probe Failure) 清零；如果没有，把 Index 寄存器的最高位 P 置 1。检查是否匹配时不能忘掉的事情有以下两个。一个是要考虑屏蔽位 (PageMask 寄存器中的 Mask)，即只比较 VPN2 中的那些没被屏蔽的位；另一个是要考虑全局标志 G。如果 G = 0，还要比较二者的 ASID。

3. tlbr (Read Indexed TLB Entry) 指令

tlbr 指令读 TLB，把由 Index 寄存器中的 Index 域指定的 TLB 项的内容送到 EntryHi、EntryLo0、EntryLo1 和 PageMask 寄存器。注意 TLB 项中的一位 G 要写入 EntryLo0 和 EntryLo1 两个寄存器中的 G 位 (相同了)，而当初写 TLB 时 EntryLo0 和 EntryLo1 两个寄存器中的 G 位可能不同，逻辑“与”后写入 TLB 项中的 G 位。

4. tlbwi (Write Indexed TLB Entry) 指令

tlbwi 指令写 TLB，把 EntryHi、EntryLo0、EntryLo1 和 PageMask 寄存器的内容送到由 Index 寄存器中的 Index 域指定的 TLB 项的相应域。注意 TLB 项中的一位 G 由 EntryLo0 和 EntryLo1 寄存器中的两位 G 逻辑“与”得到。

5. tlbwr (Write Random TLB Entry) 指令

tlbwr 指令也是写 TLB，完成与 tlbwi 类似的任务，不同点是它不使用 Index 寄存器，而是使用随机数寄存器 Random 来指定 TLB 项。

流水线 CPU 一定要有两个 TLB：指令 TLB 和数据 TLB，分别用于取指令和访问数据时的地址转换。而 MIPS 只提供了一套 TLB 读写指令，并且没有提供其他的手段能用软件分别读写这两个 TLB，导致 TLB 的设计变得有些麻烦。搞不懂设计者当初是怎么想的。

12.5 习题

1. 简要解释什么是 Cache、MMU 和 TLB 以及需要它们的原因。
2. 调查扇区映像 (Sector Mapping) 的 Cache 结构。
3. 设计并仿真一个两路组相联的 Cache 电路。注意要有与 CPU 和存储器的接口信号，其他策略自己决定。
4. 在写透策略中，不管 Cache 是否命中，每次写数据操作都要写存储器。本章给出的例子是 CPU 等待写存储器操作完成。试设计一个带有写缓冲区的 Cache，使得 CPU 不必等待。
5. 试调查虚地址 Cache 的原理与设计。

6. 试用 Verilog HDL 设计一个四路组相联的 TLB 电路并对电路进行仿真。
7. 试设计一个 CPU，使其能执行以下 MIPS 指令：mfc0、mtc0、tlbp、tlbr、tlbwi 和 tlbwr。
8. 用踪迹驱动模拟或执行驱动模拟的方法定量评价各种结构的 Cache，包括映像进制、Cache 的容量、块的大小、替换策略、写策略等。

第 13 章　带有 Cache 及 TLB 和 FPU 的 CPU 设计

本章描述一个完整的流水线 CPU 的设计，包括整数部件、浮点部件、分开的指令 Cache 和数据 Cache 以及分开的指令 TLB 和数据 TLB。浮点部件能完成加减乘除和开方运算，Cache 完全由硬件控制，而 TLB 的维护需要软件介入。

TLB 不命中时产生异常信号，因此本章给出的 CPU 也包含了异常处理。TLB 维护指令只实现 tlbwi 和 tlbwr 两条指令。另外，我们在 Index 寄存器的第 30 位增设了一位 D (Data TLB)。当 D = 1 时，tlbwi 和 tlbwr 写数据 TLB；为 0 时写指令 TLB。本章给出 CPU 的 Verilog HDL 设计代码以及仿真波形。

13.1　Cache 和 TLB 的总体结构

我们已经在第 12 章介绍了 Cache 和 TLB 的工作原理及它们的 Verilog HDL 代码。本节的重点是如何使用它们，与整数部件和浮点部件以及主存有机地结合在一起，为 CPU 提供有效的存储器层次的管理。图 13.1 是简化的 TLB 与 Cache 之间的连接概念图。

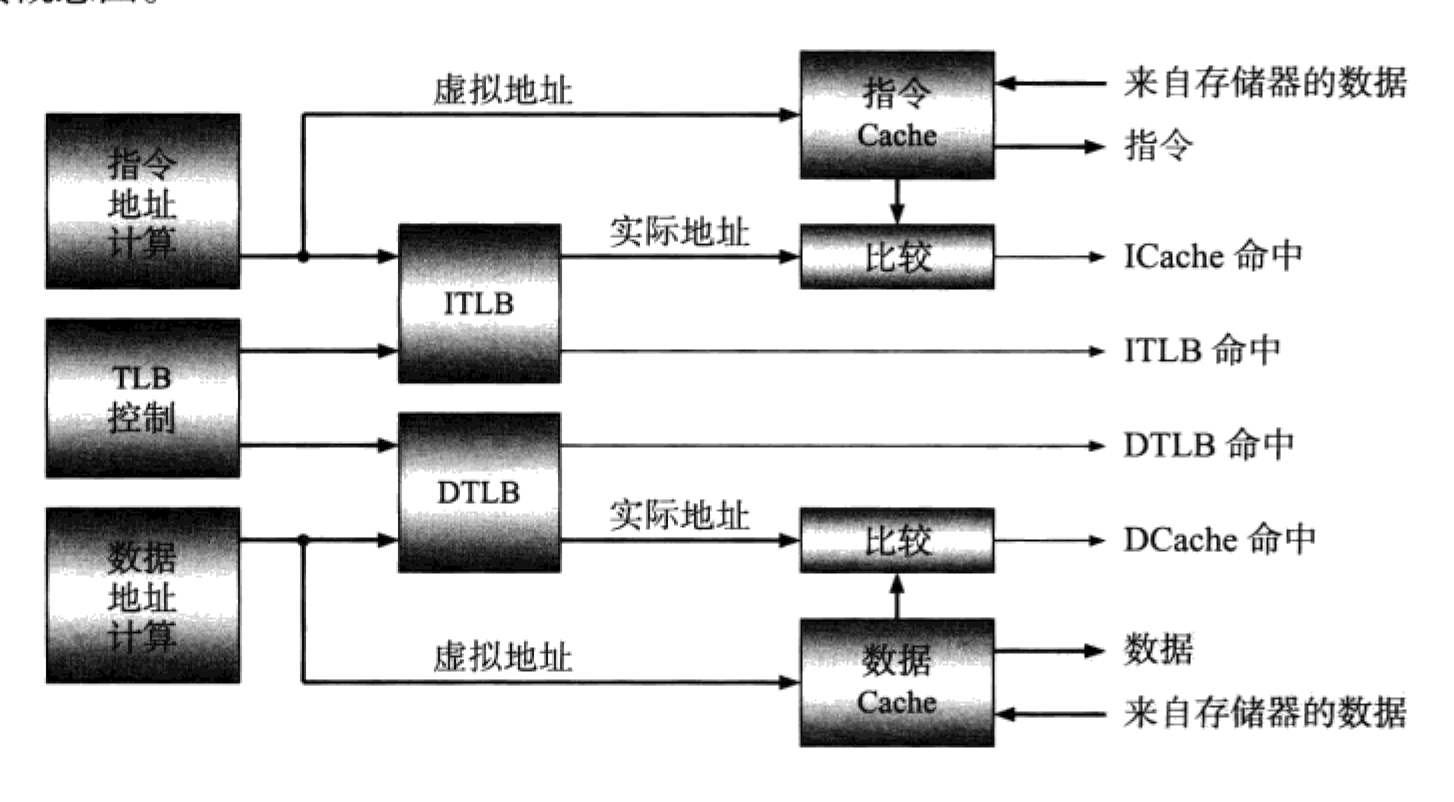


图 13.1　TLB 与 Cache 的连接

TLB 的作用是把 CPU 产生的虚拟地址快速地转换成存储器的实际地址。我们使用两个 TLB：指令 TLB (简称 ITLB) 和数据 TLB (简称 DTLB)，分别用于指令地址和数据地址的转换。为了设计简单起见，我们使用固定大小的 4KB 页面，TLB 的映像方式为全相联映像。

Cache 的作用是为 CPU 快速地提供指令和数据。我们也使用两个 Cache：指令

Cache 和数据 Cache。Cache 的映像方式为最简单的直接映像，标志位使用经过 TLB 转换过的实际地址。

13.2 与 Cache 有关的电路设计

13.2.1 指令 Cache 的 Verilog HDL 代码

我们已经在第 12 章介绍了数据 Cache 的设计方法并给出了它的 Verilog HDL 代码。指令 Cache 比数据 Cache 简单，因为 CPU 并不往指令 Cache 写任何东西。以下是指令 Cache 的 Verilog HDL 代码。请与数据 Cache 的代码进行比较。

```
module i_cache #(parameter A_WIDTH = 32, parameter C_INDEX = 6)
  (p_a,p_din, p_strobe,p_ready,cache_miss,clk,clrn,
   m_a, m_dout,m_strobe,m_ready);
    input [A_WIDTH-1:0]  p_a;
    output [31:0]        p_din;
    input                p_strobe;
    output               p_ready;
    output               cache_miss;
    input                clk, clrn;
    output [A_WIDTH-1:0] m_a;
    input [31:0]         m_dout;
    output               m_strobe;
    input                m_ready;
    localparam T_WIDTH = A_WIDTH - C_INDEX - 2; // 1 block = 1 word
    reg                  d_valid [0:(1<<C_INDEX)-1];
    reg [T_WIDTH-1:0]    d_tags  [0:(1<<C_INDEX)-1];
    reg [31:0]           d_data  [0:(1<<C_INDEX)-1];
    wire [C_INDEX-1:0]   index = p_a[C_INDEX+1:2];
    wire [T_WIDTH-1:0]   tag = p_a[A_WIDTH-1:C_INDEX+2];
    // write to cache
    always @ (posedge clk or negedge clrn)
        if (clrn == 0) begin
            integer i;
            for (i = 0; i < (1 << C_INDEX); i = i + 1)
                d_valid[i] <= 1'b0;
        end else if (c_write)
            d_valid[index] <= 1'b1;
    always @ (posedge clk)
        if (c_write) begin
            d_tags[index] <= tag;
            d_data[index] <= c_din;
        end
    // read from cache
    wire               valid = d_valid[index];
```

```
    wire [T_WIDTH-1:0] tagout = d_tags[index];
    wire [31:0]        c_dout = d_data[index];
    // cache control
    wire   cache_hit  = valid & (tagout == tag); // hit
    assign cache_miss = ~cache_hit;
    assign m_a        = p_a;
    assign m_strobe   = p_strobe & cache_miss ; // read on miss
    assign p_ready    = cache_hit | cache_miss & m_ready;
    wire   c_write    = cache_miss & m_ready;
    wire   sel_out    = cache_hit;
    wire [31:0] c_din = m_dout;
    assign      p_din = sel_out? c_dout : m_dout;
endmodule
```

13.2.2 数据 Cache 和指令 Cache 与外部存储器的接口

因为有分开的指令 Cache 和数据 Cache，CPU 从指令 Cache 取指令的同时，也可以访问数据 Cache。但与存储器的接口只有一套，见图 13.2。

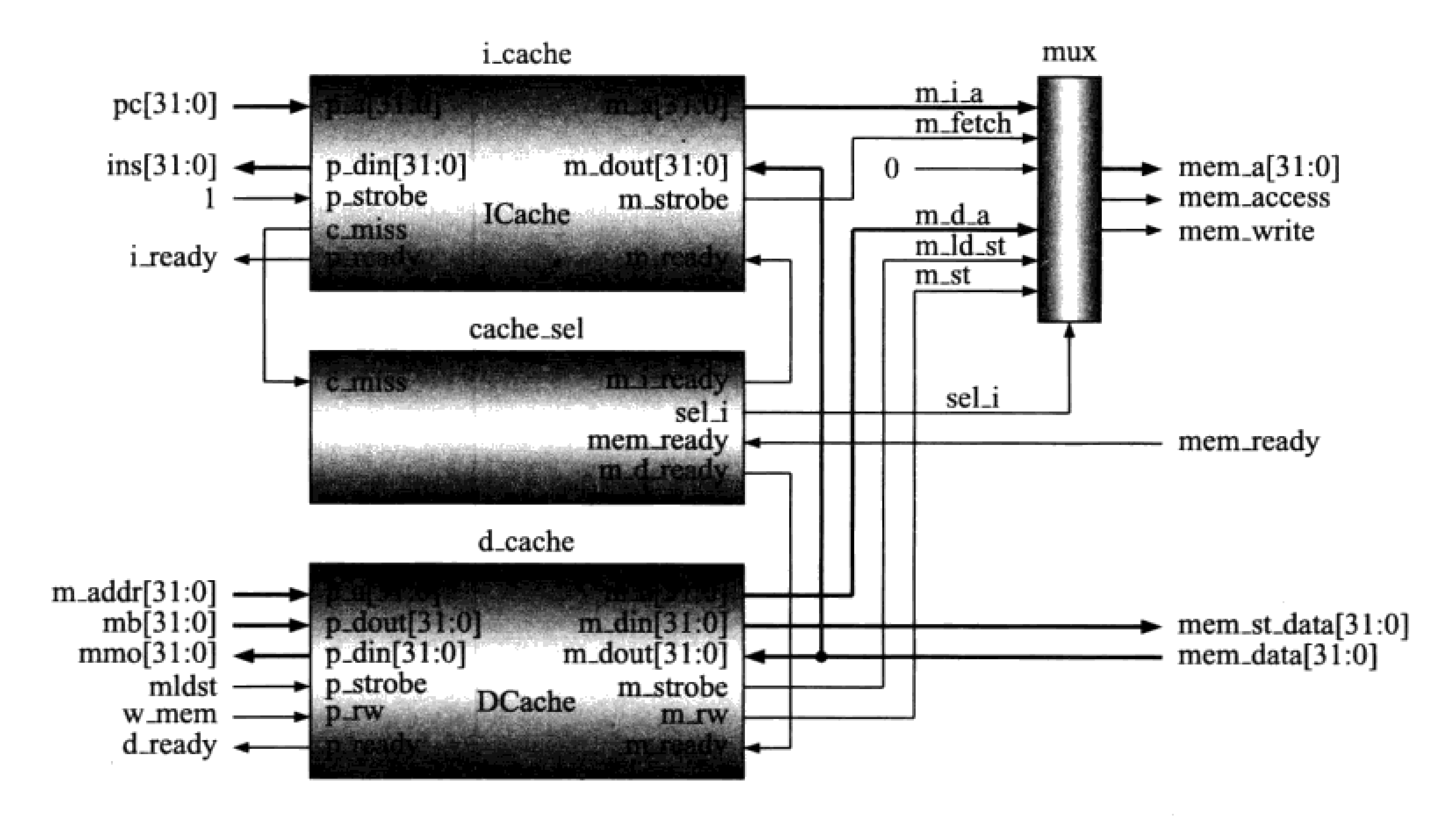


图 13.2 数据 Cache、指令 Cache 和存储器接口

图中左边的信号连接到 CPU 中的 IU 部分，右边的信号连接到主存。我们规定指令 Cache 的优先级比数据 Cache 高，即当两个 Cache 都不命中时，首先从存储器中取出指令。图中的 mux 和 cache_sel (demux) 电路的代码如下。

```
    // mux, i_cache has higher priority than d_cache
    sel_i      = i_cache_miss;
```

```
    mem_a       = sel_i ? m_i_a    : m_d_a;
    mem_access = sel_i ? m_fetch : m_ld_st;
    mem_write  = sel_i ? 1'b0     : m_st;
    // demux
    m_i_ready  = mem_ready &  sel_i;
    m_d_ready  = mem_ready & ~sel_i;
```

13.2.3 Cache 不命中时流水线暂停的电路

Cache 不命中时，要访问存储器。在访问存储器期间，我们采用最简单的处理方法：暂停流水线。以下是 Cache 没有暂停流水线的条件：

```
    no_cache_stall = ~(~i_ready | mldst & ~d_ready);
```

式中的 mldst 是流水线 MEM 级的信号，它表示当前在 MEM 级的指令是存储器访问指令。原有的流水线暂停信号 wpcir 还是控制 PC 和 IR，而 no_cache_stall 控制包括 PC 和 IR 在内的所有的流水线寄存器。

13.3 与 TLB 有关的电路设计

我们已经在第 12 章介绍了 TLB 模块本身的设计方法并给出了电路图。本节介绍如何使用它来实现对指令虚拟地址和数据虚拟地址的转换。与 Cache 的处理方法不同，当 ITLB 或 DTLB 不命中时，产生相应的异常信号，CPU 执行异常处理程序来填充 TLB。ITLB 和 DTLB 具有相同的结构，都使用 tlb_8_entry 模块。我们首先给出它的 Verilog HDL 代码，它等同于第 12 章的电路图 (见图 12.20)。

```
module tlb_8_entry (pte_in,tlbwi,tlbwr,index,vpn,memclk,clk,clrn,
                    random,pte_out,hit,vpn_index,vpn_found);
    input  [23:0] pte_in;
    input         tlbwi, tlbwr;
    input   [2:0] index;
    input  [19:0] vpn;
    input         memclk, clk, clrn;
    output  [2:0] random;
    output [23:0] pte_out; // v d c c ppn
    output        hit;
    output  [2:0] vpn_index;
    output        vpn_found;
    wire    [2:0] w_idx, ram_idx;
    wire          tlbw = tlbwi | tlbwr;
    rand3   rdm (clk,clrn,random);
    mux2x3  w_address (index,random,tlbwr,w_idx);
    mux2x3  ram_address (vpn_index,w_idx,tlbw,ram_idx);
    ram8x24 pte (ram_idx,pte_in,memclk,memclk,tlbw,pte_out);
```

```
    cam8x20 valid_tag (memclk,vpn,w_idx,tlbw,vpn_index,vpn_found);
    assign  hit = pte_out[23] & vpn_found;
endmodule
```

13.3.1 指令 TLB (ITLB) 和数据 TLB (DTLB)

图 13.3 给出了两个 TLB 及周边电路的模块图。右边的两个输出信号 ipte_out 和 dpte_out 分别是指令和数据实际存储器地址的页号。寄存器 Index 主要存放 TLB 项的号码，另外第 30 位指出是写 ITLB 还是写 DTLB (作者定义的)。EntryLo (只有一个) 主要存放存储器实际地址的页号，它的内容是从存储器页表读来的，当 TLB 不命中时被写入 TLB。

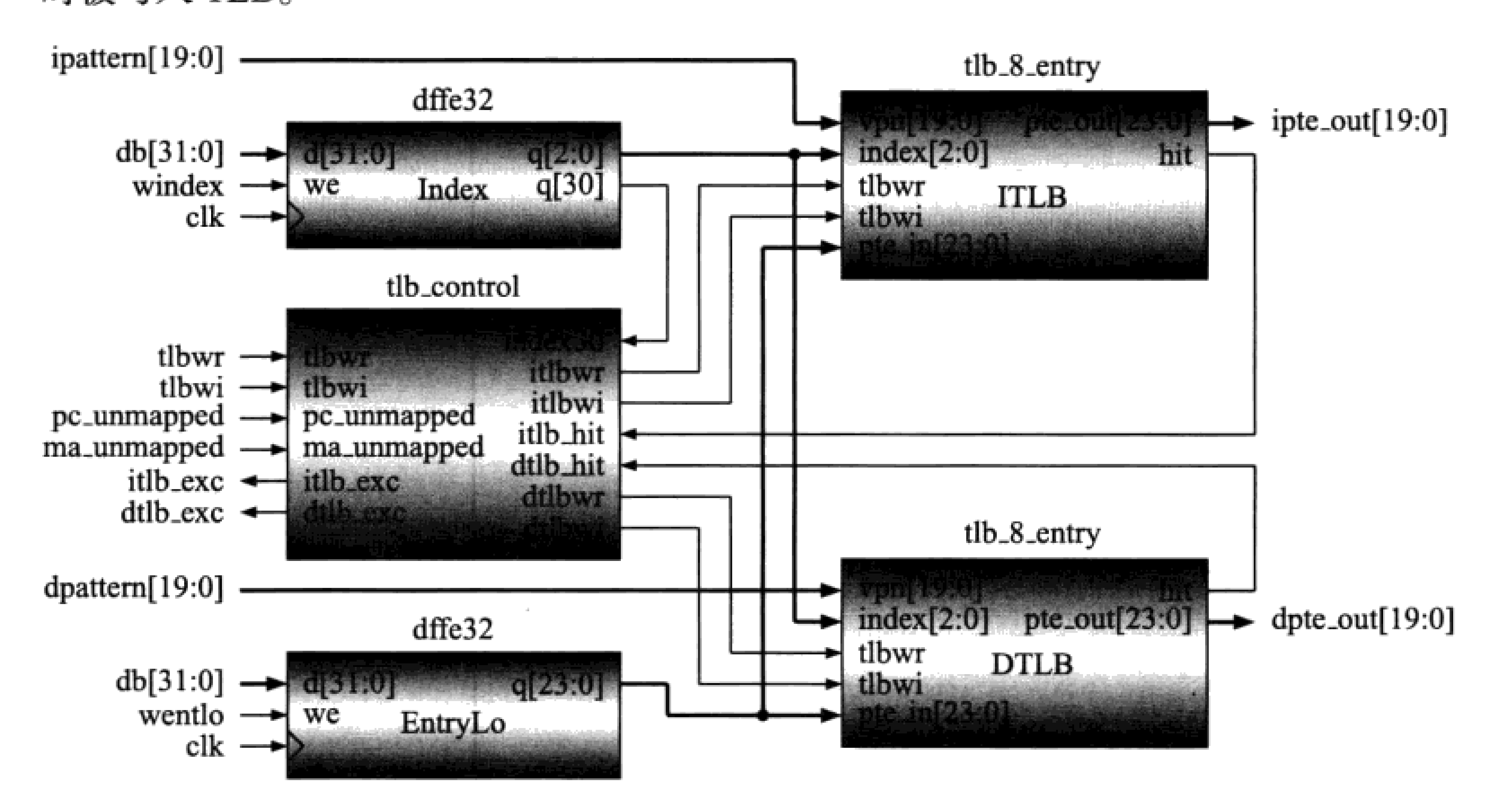


图 13.3 ITLB 和 DTLB

输入信号 ipattern 和 dpattern 各自都有两个来源：通常情况下来自于 CPU 虚拟地址的页号，用于匹配 TLB 来做地址转换；TLB 不命中时来自于 ENTRY_HI 寄存器，它的内容要被写入 TLB。ENTRY_HI 的内容实际上也是存储器虚拟地址的页号。两个信号的产生方法如下，其中 v_pc 是指令的虚拟存储器地址、malu 是数据的虚拟存储器地址。

```
    ipattern = (itlbwi | itlbwr) ? enthi[19:0] : v_pc[31:12];
    dpattern = (dtlbwi | dtlbwr) ? enthi[19:0] : malu[31:12];
```

13.3.2 TLB 不命中时异常信号的产生

并不是所有的存储器访问都要用 TLB 做地址转换。见图 12.24，当虚拟地址的最高两位为 10 时，只要把最高位的 1 改为 0，就得到实际地址。因此，当虚拟地址的最高两位为 10 时，要封锁 TLB 不命中时产生的异常信号。这部分的代码如下。

```
// mapped or unmapped
pc_unmapped = v_pc[31] & ~v_pc[30]; // 10x; v_pc: va of inst
ma_unmapped = malu[31] & ~malu[30]; // 10x; malu: va of data
// real addresses
pc     = pc_unmapped ? {1'b0,v_pc[30:0]} :
                       {ipte_out[19:0],v_pc[11:0]};
m_addr = ma_unmapped ? {1'b0,malu[30:0]} :
                       {dpte_out[19:0],malu[11:0]};
// exceptions
itlb_exc = ~itlb_hit & ~pc_unmapped;
dtlb_exc = ~dtlb_hit & ~ma_unmapped & mldst;
```

其中，itlb_exc 是 ITLB 不命中的异常信号，dtlb_exc 是 DTLB 不命中的异常信号。我们规定 itlb_exc 的优先级比 dtlb_exc 高，即当二者同时为 1 时，首先处理 dtlb_exc。

13.3.3 与 TLB 不命中异常有关的寄存器

对 TLB 不命中异常的处理要用到一些特殊的寄存器，见图 13.4。注意有些寄存器的定义与 MIPS 不同。这些寄存器的意义如下所述。

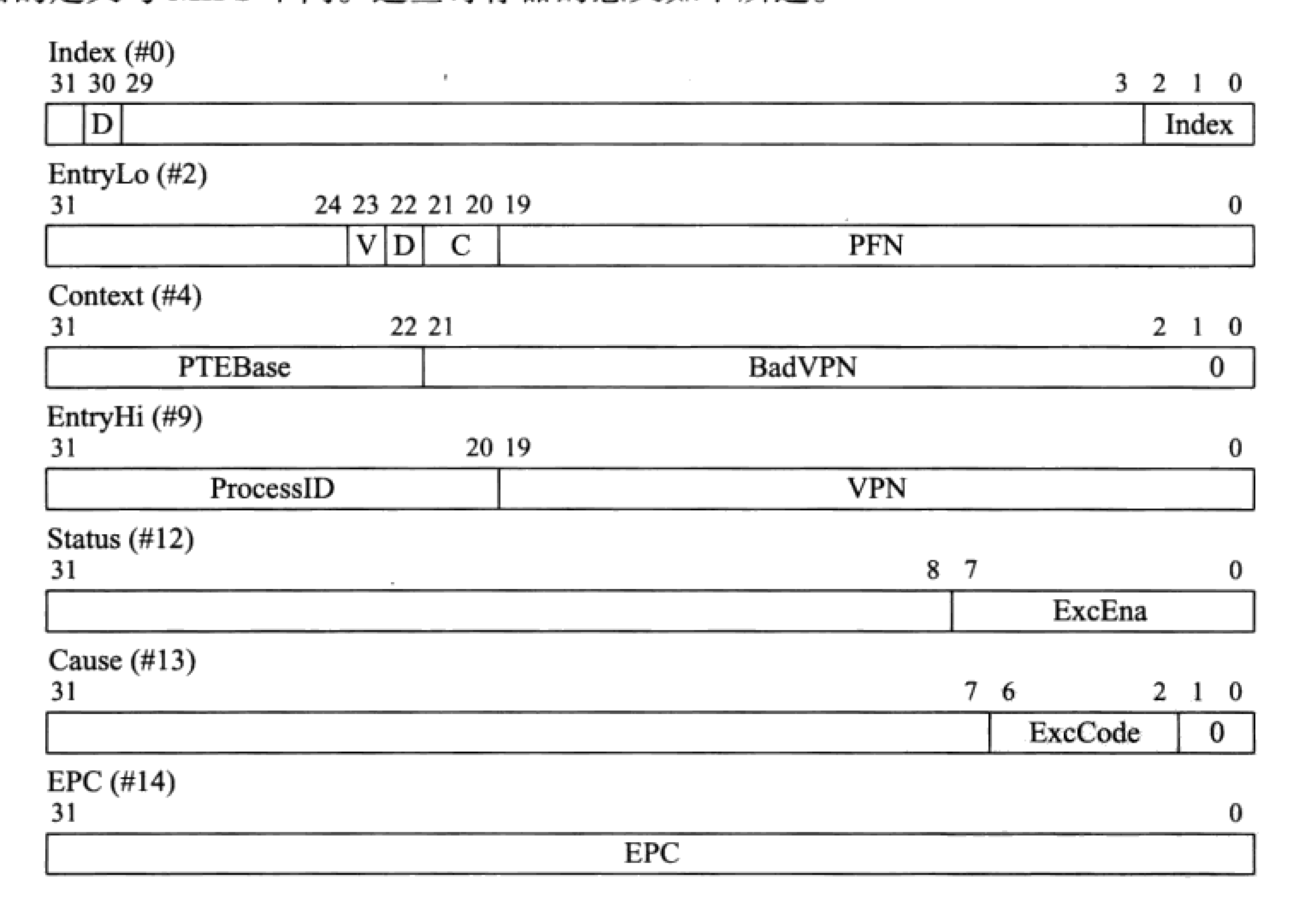


图 13.4 与 TLB 不命中异常有关的寄存器

1) CPU 通过执行 tlbwi 指令，可以修改某个 TLB 项。寄存器 Index 的最低 3 位用来指定这个 TLB 项 (在我们的设计中，TLB 总共有 8 项)。第 30 位 D 用来指定是写 ITLB 还是写 DTLB。

2) 寄存器 EntryLo 只有一个，其中的 PFN 是存储器实际地址的页号；V 是有效位；D 和 C 没有使用。
3) 寄存器 Context 的内容是一个页表项在存储器中的实际地址。当 TLB 不命中时，CPU 使用这个地址从页表中读出一个页表项，将其写入 TLB。Context 中的高位部分 PTEBase 由 CPU 执行 mtc0 指令写入；低位部分 BadVPN 由硬件自动写入，它是引起 TLB 不命中异常的虚拟地址的页号。
4) 寄存器 EntryHi 中的 VPN 是虚拟地址的页号，由 CPU 设置。CPU 执行 tlbwi 或 tlbwr 指令时，把它写入 TLB。ProcessID 没有使用。
5) 寄存器 Status 中的 ExcEna 是 8 位异常允许位，其中第 4 位和第 5 位分别对应 itlb_exc 和 dtlb_exc。为 0 时，屏蔽相应的异常。当 CPU 响应异常时，把 Status 寄存器的内容左移 8 位以屏蔽进一步的异常 (作者的做法)。
6) 寄存器 Cause 中的 ExcCode 指出当前发生的是哪种异常：itlb_exc 和 dtlb_exc 的 ExcCode 分别定义为 4 和 5。
7) 寄存器 EPC 用来保存异常返回地址，由硬件自动写入。

以下描述 ITLB 和 DTLB 不命中产生异常时硬件需要完成的动作。图 13.5 所示的是在通常情况下出现 itlb_exc 时的流水线时序和保存返回地址的电路。

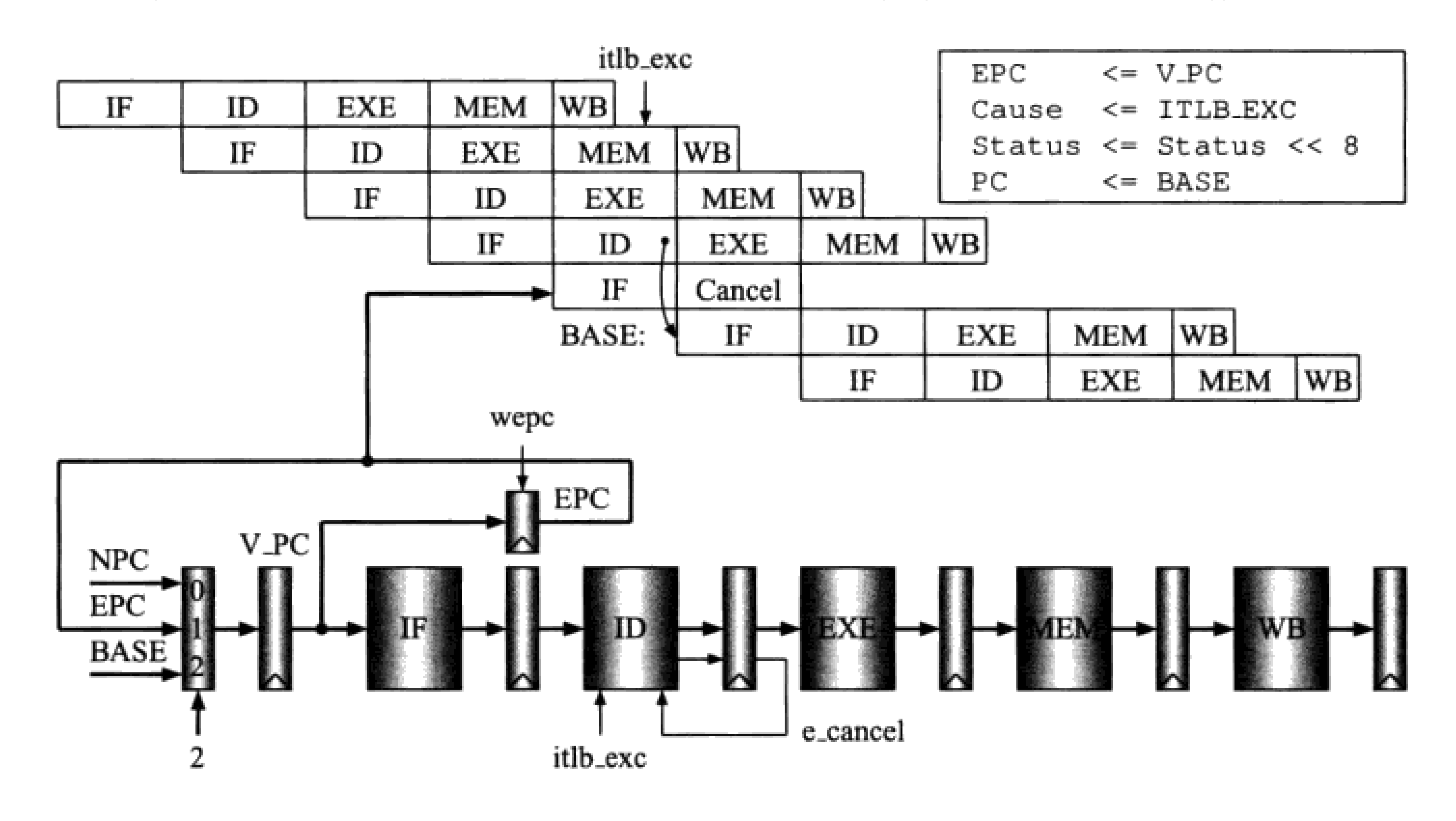


图 13.5 一般情况下的 ITLB_EXC

ITLB 不命中异常 itlb_exc 出现在取指令的 IF 级。此时的 ID 级要对该异常进行处理。处理动作包括：(1) 把引起异常的指令的虚拟地址保存到 EPC；(2) 把 4 写入 Cause 寄存器的 ExcCode 域；(3) Status 寄存器左移 8 位；(4) 把异常处理程序的入口地址写入 V_PC；(5) 产生废弃指令的信号 cancel。以上五个动作同时完成。

因为已经把异常处理程序的入口地址写入了 V_PC，所以 CPU 将执行程序以实现对 ITLB 的修改。具体的做法稍后讨论。我们还是接着讲 ITLB 和 DTLB 异常。

图 13.6 所示的是在取延迟槽指令的情况下出现 itlb_exc 时的流水线时序和保存返回地址到 EPC 的电路。此时处在 ID 级的指令是转移指令，本来好好地要转移到目标地址，但由于出现了异常，必须要转移到异常处理程序的入口，因此保存到 EPC 的返回地址必须是转移指令的地址，即返回后重新执行转移指令。注意此时的 V_PC 指向的是延迟槽指令。为了能够得到转移指令的地址，我们增加了一个流水线寄存器 PCD。此时 PCD 中的内容就是转移指令的地址，把它保存到 EPC 中就行了。

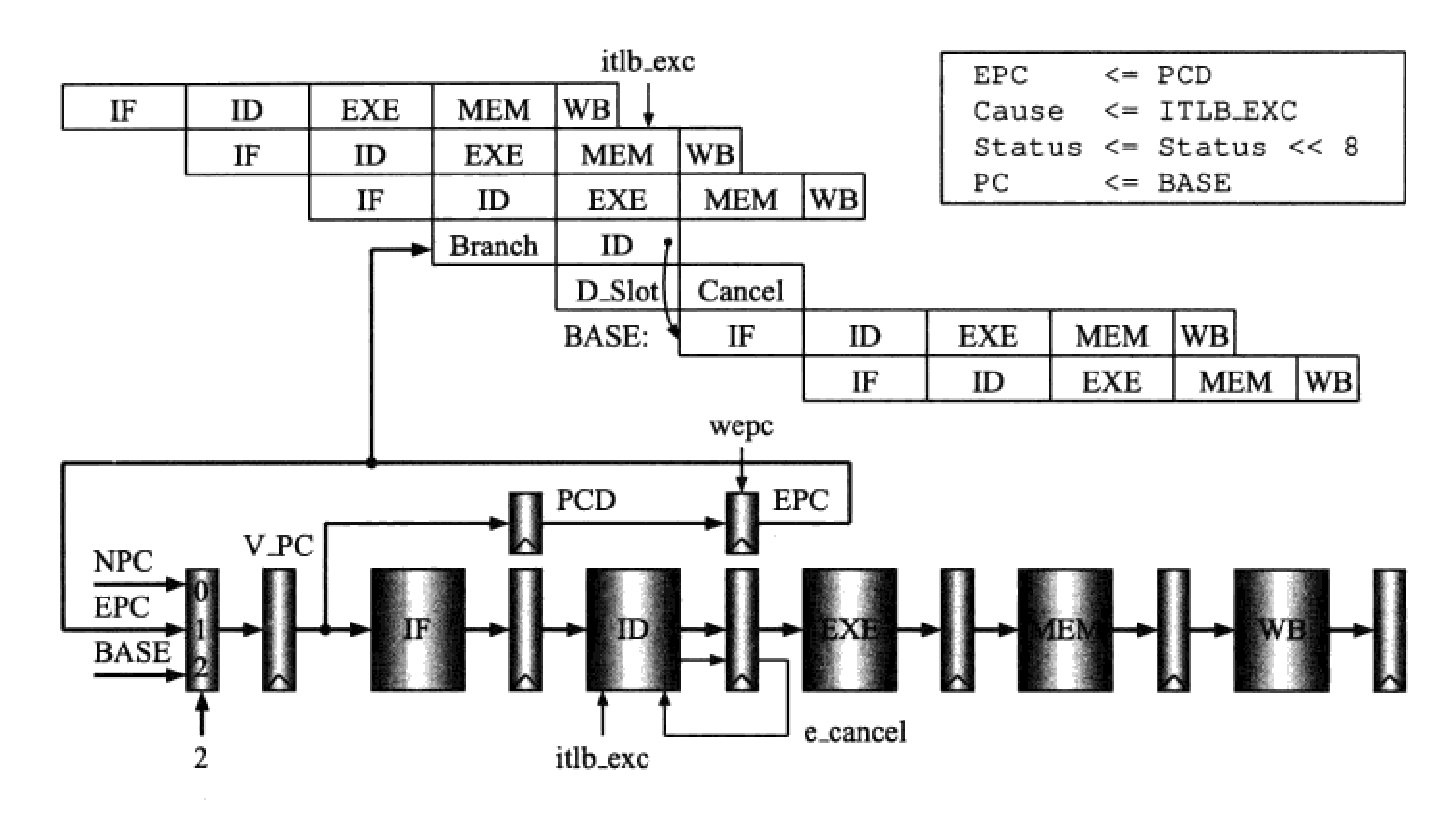


图 13.6 处在延迟槽的指令引起的 ITLB_EXC

DTLB 出现不命中异常就有一点点小麻烦了，它出现在流水线的 MEM 级，与 ITLB 的不命中异常出现在 IF 级相比，整整晚了 3 级。倒不是说保存返回地址有多麻烦，真正麻烦的是要废弃好多指令。

图 13.7 所示的是在通常情况下出现 dtlb_exc 时的流水线时序和保存返回地址的电路。要废弃的指令总共有 4 条，它们分别处在 IF、ID、EXE 和 MEM 级。废弃信号在 ID 级产生，“自废”没问题，但除此之外还要忙前忙后。废弃下一条 IF 级的指令也没问题，因为已经在 ITLB 不命中时演练过了。废弃 EXE 级指令的办法是把 EXE 级的控制信号在写入流水线寄存器之前封锁掉 (指令在 EXE 级不改变 CPU 状态)。废弃 MEM 级的指令时，不仅要把写入流水线寄存器的控制信号封锁掉，也要把在 MEM 级使用的写存储器信号封锁掉。写入 EPC 的返回地址在 PCM 中，即引起 DTLB 不命中异常的指令的地址。

图 13.8 所示的是处在延迟槽的指令引起 dtlb_exc 时的流水线时序和保存返回地址的电路。与 ITLB 的情况类似，返回地址应该是转移指令的地址，它在 PCW 寄存器中。废弃指令的动作与上述一般情况下的废弃指令的动作相同。

我们把以上 4 种情况加以总结，列在表 13.1 中。由此我们得到选择信号 sepc 的逻辑表达式如下。它选择不同的返回地址，选中的地址被写入 EPC 中。

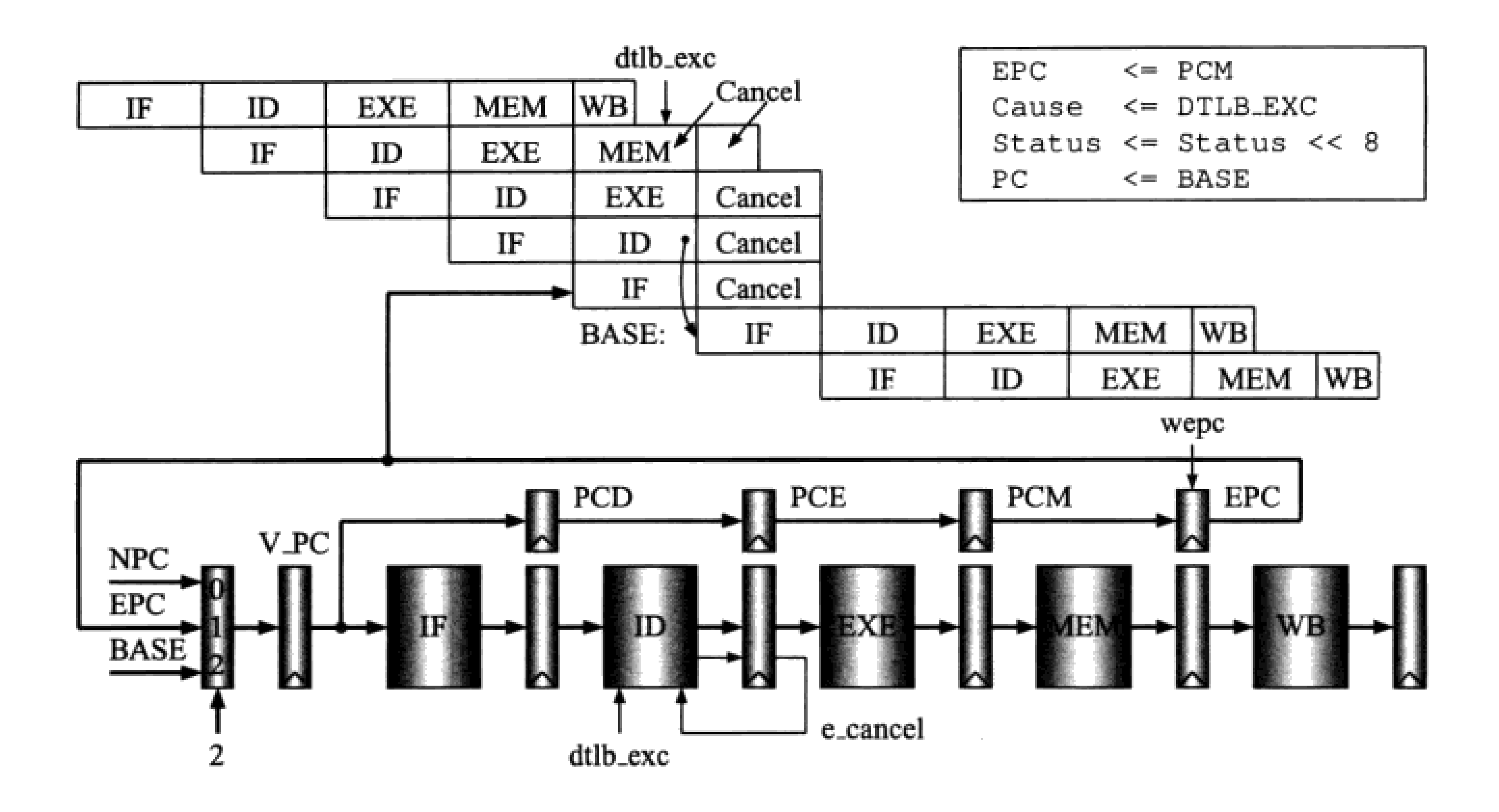


图 13.7 一般情况下的 DTLB_EXC

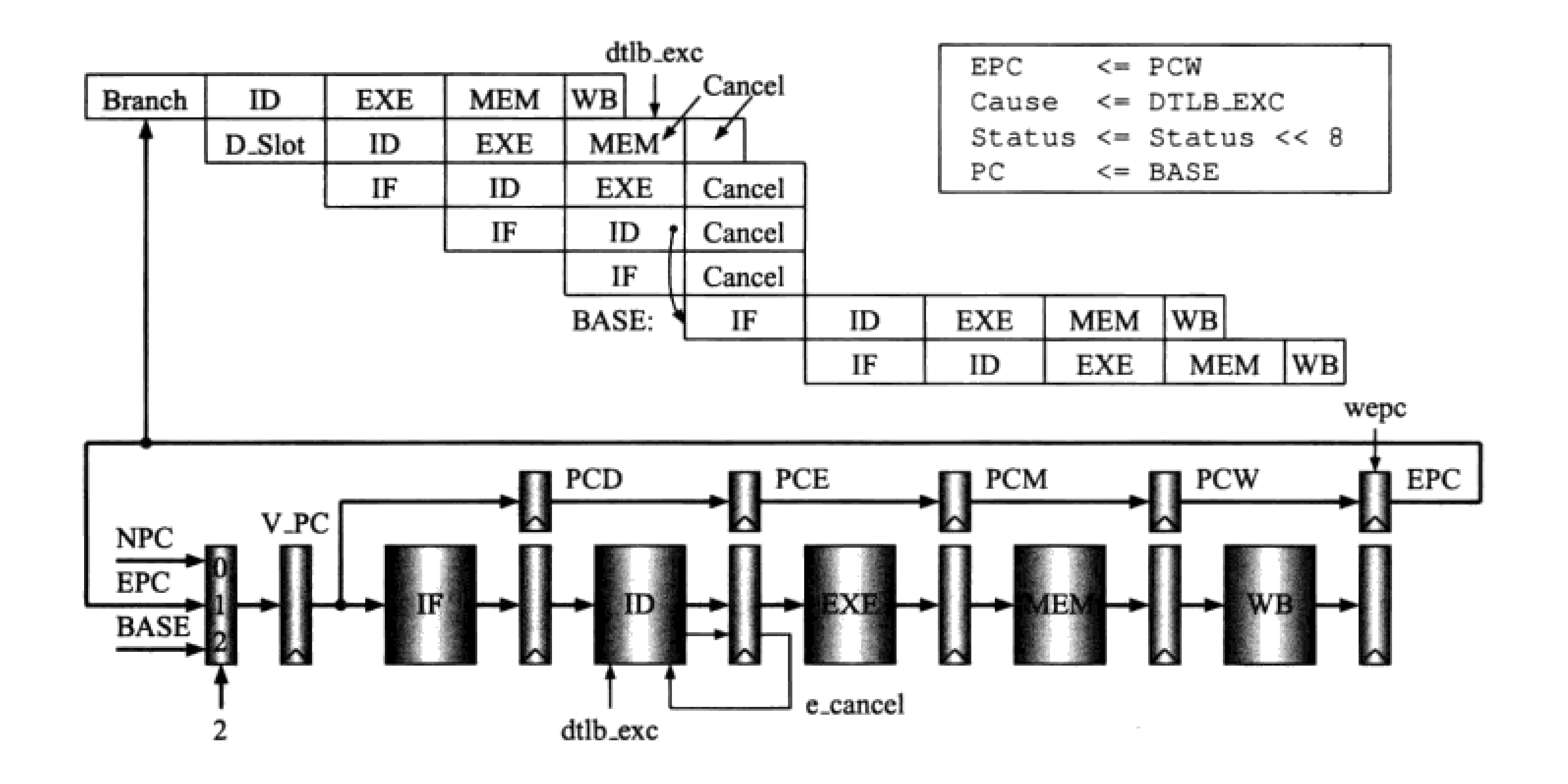


图 13.8 处在延迟槽的指令引起的 DTLB_EXC

表 13.1 为 EPC 选择返回地址

itlb_exc	dtlb_exc	isbr	wisbr	EPC	sepc[1:0]
1	x	0	x	V_PC	0 0
1	x	1	x	PCD	0 1
0	1	x	0	PCM	1 0
0	1	x	1	PCW	1 1

```
sepc[1] = ~itlb_exc & dtlb_exc;
sepc[0] =  itlb_exc & isbr | ~itlb_exc & dtlb_exc & wisbr;
```

13.3.4 对 TLB 不命中异常的处理

当 TLB 不命中异常发生且异常没被屏蔽时，CPU 跳转到固定的地址 (BASE) 去执行异常处理程序。在我们的设计中，这个地址是 0x80000008。当 CPU 处理完异常，执行 eret 指令时，EPC 的内容要写入程序计数器中，以便实现从异常处理程序返回。因此我们在程序计数器的输入端加了一个多路器。多路器的输入及输入选择信号如下。其中 exc 是异常信号，i_eret 为 1 表示当前指令是 eret。

```
// selpc[1:0]:  00: npc;  01: epc;  10: EXC_BASE
selpc[1] = exc;
selpc[0] = i_eret;
```

当异常发生时，由 CPU 硬件将异常源写入 Cause 寄存器的 ExcCode 域。如前所述，我们把 itlb_exc 和 dtlb_exc 的 ExcCode 分别定义为 4 和 5。当 CPU 进入异常处理程序后，可根据这个 ExcCode 转入相应的程序去处理。我们的做法是设置一个跳转表 (j_table)，把 ExcCode 作为地址偏移量从跳转表得到入口地址，然后跳转到这个地址。以下的汇编程序演示这个过程 (程序中的 EXC_BASE = BASE)。

```
EXC_BASE:                               # exception/interrupt entry
0x80000008: mfc0 r26, C0_CAUSE          # read cp0 Cause reg
0x8000000c: andi r26, r26, 0x1c         # get ExcCode, 3 bits here
0x80000010: lui  r27, 0x8000            # j_table address high
0x80000014: or   r27, r27, r26          # j_table address low
0x80000018: lw   r27, j_table(r27)      # get address from table
0x8000001c: nop                         #
0x80000020: jr   r27                    # jump to that address
0x80000024: nop                         #

.data
j_table:              # address table for exception and interrupt
0x80000040: 0x80000030 # 0. int_entry, addr. for interrupt
0x80000044: 0x8000003c # 1. sys_entry, addr. for Syscall
0x80000048: 0x80000054 # 2. uni_entry, addr. for Unimpl. inst.
0x8000004c: 0x80000068 # 3. ovf_entry, addr. for Overflow
0x80000050: 0x800000c0 # 4. itlb_entry, addr. for itlb miss
0x80000054: 0x80000140 # 5. dtlb_entry, addr. for dtlb miss
0x80000058: 0x80000000 # 6.
0x8000005c: 0x80000000 # 7.
```

比如当前的异常是 dtlb_exc，则跳转到地址 0x80000140。处理 dtlb_exc 异常的主要工作是为没命中的虚拟地址在 DTLB 中准备一个 TLB 项，填上实际地址的页号。

该页号从页表中得到。修改 TLB 的指令有两条：tlbwi 和 tlbwr。我们的演示程序使用了 tlbwi，该方法需要有一个 index 号码，用来指出写哪个 TLB 项。为此我们准备了一个计数器，每次写 TLB 时，都把它加 1 (用软件实现先进先出替换策略)。由于我们的 TLB 只有 8 项，因此只使用计数器的低 3 位。这 3 位计数器值连同 DTLB 标志 D 一起写入 Index 寄存器。寄存器 Context 中的内容实际上就是页表的存储器地址，我们可以从该地址得到实际地址的页号，把它写入 EntryLo 寄存器。我们还要设置 EntryHi 寄存器，它的内容应该是引起 DTLB 不命中的虚拟地址的页号。以上寄存器都设置好之后，执行 tlbwi 指令修改 TLB，然后返回。这部分程序如下。

```
0x80000140: lui  r27, 0x8000       # 0x800001fc: counter
0x80000144: lw   r26, 0x1fc(r27)  # load dtlb index counter
0x80000148: addi r26, r26, 1      # index + 1
0x8000014c: andi r26, r26, 7      # 3-bit index
0x80000150: sw   r26, 0x1fc(r27)  # store index
0x80000154: lui  r27, 0x4000       # dtlb tag D (bit 30)
0x80000158: or   r26, r27, r26    # dtlb tag and index
0x8000015c: mtc0 r26, C0_INDEX     # move to c0 index
0x80000160: mfc0 r27, C0_CONTEXT   # move from c0 context
0x80000164: lw   r26, 0x0(r27)    # get pte
0x80000168: mtc0 r26, C0_ENTRY_LO # move to c0 entry_lo
0x8000016c: sll  r26, r27, 10     # get bad vpn
0x80000170: srl  r26, r26, 12     # for c0 entry_hi
0x80000174: mtc0 r26, C0_ENTRY_HI # move to entry_hi
0x80000178: tlbwi                 # update dtlb
0x8000017c: eret                  # return from exception
0x80000180: nop                   #
```

13.4 带有 Cache 及 TLB 的 CPU 设计

13.4.1 带有 Cache 及 TLB 的 CPU 总体结构

图 13.9 给出的是带有 Cache 及 TLB 的 CPU 总体结构的示意图。图中的 IU/FPU 模块的基本结构与第 10 章描述的 CPU 相同，但增加了对 TLB 不命中异常的处理电路，包括跳转到异常处理程序及从中返回的电路、与 TLB 有关的寄存器和对相关指令的译码电路等。

与第 10 章的 CPU 不同的是当 CPU 复位时，程序计数器的初始值为 0x80000000，而不是 0x00000000。另外，为了测试 TLB 不命中异常，我们在测试程序中扩大了数据存储器的使用范围。为此，我们使用了不连续的 4 段物理存储器，它们分别存放：(1) 系统复位时的初始化程序和异常处理程序；(2) 用于虚拟存储器管理的页表；(3) 用于测试 IU/FPU 的用户程序；(4) 用户程序所使用的数据。以下给出 CPU 的 Verilog HDL 源代码。

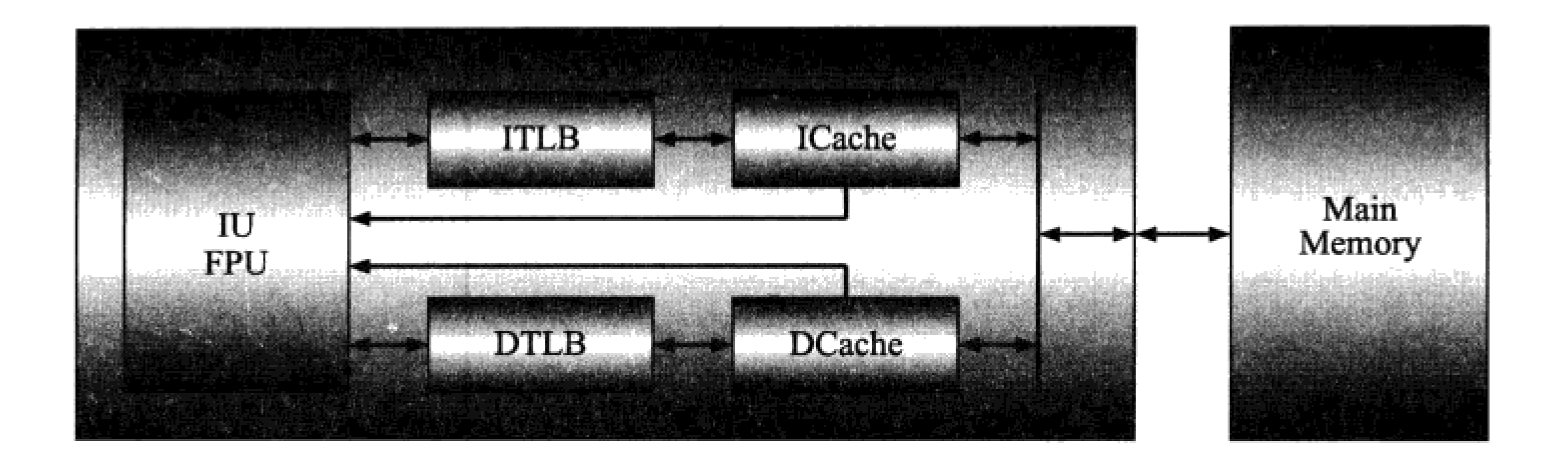


图 13.9 带有 Cache 及 TLB 的 CPU 模块图

13.4.2 带有 Cache 及 TLB 的 CPU 的 Verilog HDL 代码

以下是最顶层模块 cpu_cache_tlb_memory，包括 CPU 和存储器。

```
module cpu_cache_tlb_memory (
    clock,memclock,resetn,v_pc,pc,inst,ealu,malu,walu,wn,wd,ww,
    stall_lw,stall_fp,stall_lwc1,stall_swc1,stall,
    mem_a,mem_data,mem_st_data,mem_access,mem_write,mem_ready);
    input  clock,memclock,resetn;
    output [31:0] v_pc,pc,inst,ealu,malu,walu;
    output [31:0] wd;
    output  [4:0] wn;
    output ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,stall;
    output [31:0] mem_a;
    output [31:0] mem_data;
    output [31:0] mem_st_data;
    output        mem_access;
    output        mem_write;
    output        mem_ready;
    // cpu
    cpu_cache_tlb cpucachetlb (
       clock,memclock,resetn,v_pc,pc,inst,ealu,malu,walu,wn,wd,
       ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,stall,mem_a,
       mem_data,mem_st_data,mem_access,mem_write,mem_ready);
    // main memory
    physical_memory mem (mem_a,mem_data,mem_st_data,mem_access,
                         mem_write,mem_ready,clock,memclock,resetn);
endmodule
```

以下是 CPU 模块 cpu_cache_tlb，包括 IU、Cache、TLB 和 FPU。

```
module cpu_cache_tlb
   (clock,memclock,resetn,v_pc,pc,inst,ealu,malu,walu,wn,wd,ww,
    stall_lw,stall_fp,stall_lwc1,stall_swc1,stall,
    mem_a,mem_data,mem_st_data,mem_access,mem_write,mem_ready);
```

```
    input  clock,memclock,resetn;
    output [31:0] v_pc,pc,inst,ealu,malu,walu;
    output [31:0] wd;
    wire   [31:0] e3d;
    output  [4:0] wn;
    wire    [4:0] e1n,e2n,e3n;
    output ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,stall;
    wire   e; // for multithreading CPU, not used here
    wire    [4:0] count_div,count_sqrt; // for testing
    output [31:0] mem_a;
    input  [31:0] mem_data;
    output [31:0] mem_st_data;
    output        mem_access;
    output        mem_write;
    input         mem_ready;
    wire [31:0] qfa,qfb,fa,fb,dfa,dfb,mmo,wmo; // for iu
    wire [4:0]  fs,ft,fd;
    wire [2:0]  fc;
    wire        fwdla,fwdlb,fwdfa,fwdfb,wf,fasmds;
    wire        e1w,e2w,e3w,wwfpr;
    wire        no_cache_stall;
    iu_cache_tlb i_u (e1n,e2n,e3n, e1w,e2w,e3w, stall,1'b0,
        dfb,e3d, clock,memclock,resetn,no_cache_stall,
        fs,ft,wmo,wrn,wwfpr,mmo,fwdla,fwdlb,fwdfa,fwdfb,fd,fc,wf,
        fasmds,v_pc,pc,inst,ealu,malu,walu,
        stall_lw,stall_fp,stall_lwc1,stall_swc1,mem_a,
        mem_data,mem_st_data,mem_access,mem_write,mem_ready);
    wire [4:0] wrn;
    regfile2w fpr (fs,ft,wd,wn,ww,wmo,wrn,wwfpr,~clock,resetn,
                   qfa,qfb);
    mux2x32 fwd_f_load_a (qfa,mmo,fwdla,fa);
    mux2x32 fwd_f_load_b (qfb,mmo,fwdlb,fb);
    mux2x32 fwd_f_res_a  (fa,e3d,fwdfa,dfa);
    mux2x32 fwd_f_res_b  (fb,e3d,fwdfb,dfb);
    wire [1:0] e1c,e2c,e3c; // for fpu
    fpu fp_unit (dfa,dfb,fc,wf,fd,no_cache_stall,clock,resetn,
                 e3d,wd,wn,ww,stall,e1n,e1w,e2n,e2w,e3n,e3w,
                 e1c,e2c,e3c,count_div,count_sqrt,e);
endmodule
```

以下是 IU 模块 iu_cache_tlb，包括 IU、Cache 和 TLB。

```
module iu_cache_tlb (
    e1n,e2n,e3n, e1w,e2w,e3w, stall,st,
    dfb,e3d, clock,memclock,resetn,no_cache_stall,
    fs,ft,wmo,wrn,wwfpr,mmo,fwdla,fwdlb,fwdfa,fwdfb,fd,fc,wf,fasmds,
```

```
v_pc,pc,inst,ealu,malu,walu,
stall_lw,stall_fp,stall_lwc1,stall_swc1,
mem_a,mem_data,mem_st_data,mem_access,mem_write,mem_ready);
input [31:0] dfb,e3d;
input [4:0] e1n,e2n,e3n;
input       e1w,e2w,e3w, stall,st, clock,memclock,resetn;
output      no_cache_stall;
output [31:0] v_pc,pc,inst,ealu,malu,walu;
output [31:0] mmo,wmo;
output [4:0]  fs,ft,fd,wrn;
output [2:0]  fc;
output        wwfpr,fwdla,fwdlb,fwdfa,fwdfb,wf,fasmds;
output        stall_lw,stall_fp,stall_lwc1,stall_swc1;
output [31:0] mem_a;
input  [31:0] mem_data;
output [31:0] mem_st_data;
output        mem_access;
output        mem_write;
input         mem_ready;
parameter   EXC_BASE = 32'h80000008; // = base = BASE
wire [31:0] bpc,jpc,npc,pc4,ins,dpc4,inst,qa,qb,da,db,dimm,dc,dd;
wire [31:0] simm,epc8,alua,alub,ealu0,ealu1,ealu,sa,eb,mmo,wdi;
wire [5:0]  op,func;
wire [4:0]  rs,rt,rd,fs,ft,fd,drn,ern;
wire [3:0]  aluc;
wire [1:0]  pcsource,fwda,fwdb;
wire        wpcir;
wire        wreg,m2reg,wmem,aluimm,shift,jal;
wire [31:0] qfa,qfb,fa,fb,dfa,dfb,efb,e3d;
wire [4:0]  e1n,e2n,e3n,wn;
wire [2:0]  fc;
wire [1:0]  e1c,e2c,e3c;
reg ewfpr,ewreg,em2reg,ewmem,ejal,efwdfe,ealuimm,eshift;
reg mwfpr,mwreg,mm2reg,mwmem;
reg wwfpr,wwreg,wm2reg;
reg [31:0] epc4,ea,ed,eimm,malu,mb,wmo,walu;
reg [4:0] ern0,mrn,wrn;
reg [3:0] ealuc;
// IF
p_c vpc (next_pc,clock,resetn,wpcir&no_cache_stall,v_pc); // VPC
cla32 pc_plus4 (v_pc,32'h4,1'b0,pc4); // VPC+4
mux4x32 nextpc (pc4,bpc,da,jpc,pcsource,npc); // Next PC
wire tlbwi,tlbwr;
wire itlbwi = tlbwi & ~index[30]; // itlb write
wire itlbwr = tlbwr & ~index[30];
wire dtlbwi = tlbwi &  index[30]; // dtlb write
wire dtlbwr = tlbwr &  index[30];
```

```
wire [19:0] ipattern = (itlbwi | itlbwr) ? enthi[19:0] : v_pc[31:12];
wire pc_unmapped = v_pc[31] & ~v_pc[30]; // 10x
assign pc = pc_unmapped?{1'b0,v_pc[30:0]}:{ipte_out[19:0],v_pc[11:0]};
wire  [2:0] irandom;
wire [23:0] ipte_out;
wire        itlb_hit;
wire  [2:0] ivpn_index;
wire        ivpn_found;
tlb_8_entry itlb (entlo[23:0], itlbwi, itlbwr, index[2:0], ipattern,
                  memclock, clock, resetn,
                  irandom, ipte_out, itlb_hit, ivpn_index, ivpn_found);
wire itlb_exc = ~itlb_hit & ~pc_unmapped;
wire      i_ready,i_cache_miss;
i_cache icache (pc,ins,1'b1,i_ready,i_cache_miss,clock,resetn,
                m_i_a,mem_data,m_fetch,m_i_ready);
// IF-ID pipeline registers
dffe32 pc_4_r (pc4, clock,resetn,wpcir&no_cache_stall,dpc4); // PC4
dffe32 inst_r (ins, clock,resetn,wpcir&no_cache_stall,inst); // IR
dffe32 pcd_r  (v_pc,clock,resetn,wpcir&no_cache_stall,pcd);  // PCD
wire [31:0] pcd;
// ID
assign op   = inst[31:26];
assign rs   = inst[25:21];
assign rt   = inst[20:16];
assign rd   = inst[15:11];
assign ft   = inst[20:16];
assign fs   = inst[15:11];
assign fd   = inst[10:6];
assign func = inst[5:0];
assign simm = {{16{sext&inst[15]}},inst[15:0]};
assign jpc  = {dpc4[31:28],inst[25:0],2'b00}; // jump target
cla32 br_addr (dpc4,{simm[29:0],2'b00},1'b0,bpc); // branch target
regfile rf (rs,rt,wdi,wrn,wwreg,~clock,resetn,qa,qb); // reg file
mux4x32 alu_a (qa,ealu,malu,mmo,fwda,da); // forward A
mux4x32 alu_b (qb,ealu,malu,mmo,fwdb,db); // forward B
wire swfp,regrt,sext,fwdf,fwdfe,wfpr;
mux2x32 store_f (db,dfb,swfp,dc); // swc1
mux2x32 fwd_f_d (dc,e3d,fwdf,dd); // forward fp result
wire rsrtequ = ~|(da^db);         // rsrtequ = (da == db)
mux2x5 des_reg_no (rd,rt,regrt,drn); // destination reg
wire wepc,wcau,wsta,isbr,cancel,exc,ldst;
wire [1:0] sepc,selpc;
control cu (op,func,rs,rt,rd,fs,ft,rsrtequ,            // control unit
            ewfpr,ewreg,em2reg,ern,                    // from iu
            mwfpr,mwreg,mm2reg,mrn,                    // from iu
            e1w,e1n,e2w,e2n,e3w,e3n,stall,st,          // from fpu
            pcsource,wpcir,wreg,m2reg,wmem,jal,aluc,   // iu
```

```
            sta,aluimm,shift,sext,regrt,fwda,fwdb,      // iu
            swfp,fwdf,fwdfe,wfpr,                       // used by iu
            fwdla,fwdlb,fwdfa,fwdfb,fc,wf,fasmds,       // used by fpu
            stall_lw,stall_fp,stall_lwc1,stall_swc1,    // for testing
            windex,wentlo,wcontx,wenthi,rc0,wc0,tlbwi,tlbwr, // for tlb
            c0rn,wepc,wcau,wsta,isbr,sepc,cancel,cause,exc,selpc,ldst,
            wisbr,ecancel,itlb_exc,dtlb_exc);
wire [31:0] index; // cp0 reg 0: index
wire [31:0] entlo; // cp0 reg 2: entry lo
reg  [31:0] contx; // cp0 reg 4: context
wire [31:0] enthi; // cp0 reg 9: entry hi
wire        windex,wentlo,wcontx,wenthi; // write enables
wire        rc0,wc0; // read,write c0 res
wire [1:0]  c0rn;    // c0 reg # for mux
dffe32 c0_Index (db,clock,resetn,windex&no_cache_stall,index); // index
dffe32 c0_Entlo (db,clock,resetn,wentlo&no_cache_stall,entlo); // entlo
dffe32 c0_Enthi (db,clock,resetn,wenthi&no_cache_stall,enthi); // enthi
always @(negedge resetn or posedge clock)                      // contx
    if (resetn == 0) begin
        contx <= 0;
    end else begin
        if (wcontx) contx[31:22] <= db[31:22];              // PTEBase
        if (itlb_exc) contx[21:0] <= {v_pc[31:12],2'b00}; // BadVPN
        else if (dtlb_exc) contx[21:0] <= {malu[31:12],2'b00};
    end
wire [31:0] sta,cau,epc,sta_in,cau_in,epc_in, // for exception
            stalr,epcin,epc10,cause,c0reg,next_pc;
dffe32  c0_Status (sta_in,clock,resetn,wsta&no_cache_stall,sta); // sta
dffe32  c0_Cause  (cau_in,clock,resetn,wcau&no_cache_stall,cau); // cau
dffe32  c0_EPC    (epc_in,clock,resetn,wepc&no_cache_stall,epc); // epc
mux2x32 sta_mx (stalr,db,wc0,sta_in); // mux for Status reg
mux2x32 cau_mx (cause,db,wc0,cau_in); // mux for Cause reg
mux2x32 epc_mx (epcin,db,wc0,epc_in); // mux for EPC reg
mux2x32 sta_lr ({8'h0,sta[31:8]},{sta[23:0],8'h0},exc,stalr);
mux4x32 epc_04 (v_pc,pcd,pcm,pcw,sepc,epcin); // epc source
mux4x32 irq_pc (npc,epc,EXC_BASE,32'h0,selpc,next_pc); // for PC
mux4x32 fromc0 (contx,sta,cau,epc,ec0rn,c0reg);  // for mfc0
// ID-EXE pipeline registers
reg [31:0] pce;
reg  [1:0] ec0rn;
reg        erc0,ecancel,eisbr,eldst;
always @(negedge resetn or posedge clock)
    if (resetn == 0) begin
        ewfpr   <= 0;           ewreg   <= 0;
        em2reg  <= 0;           ewmem   <= 0;
        ejal    <= 0;           ealuimm <= 0;
        efwdfe  <= 0;           ealuc   <= 0;
```

```
            eshift  <= 0;           epc4    <= 0;
            ea      <= 0;           ed      <= 0;
            eimm    <= 0;           ern0    <= 0;
            erc0    <= 0;           ec0rn   <= 0;
            ecancel <= 0;           eisbr   <= 0;
            pce     <= 0;           eldst   <= 0;
        end else if (no_cache_stall) begin
            ewfpr   <= wfpr;        ewreg   <= wreg;
            em2reg  <= m2reg;       ewmem   <= wmem;
            ejal    <= jal;         ealuimm <= aluimm;
            efwdfe  <= fwdfe;       ealuc   <= aluc;
            eshift  <= shift;       epc4    <= dpc4;
            ea      <= da;          ed      <= dd;
            eimm    <= simm;        ern0    <= drn;
            erc0    <= rc0;         ec0rn   <= c0rn;
            ecancel <= cancel;      eisbr   <= isbr;
            pce     <= pcd;         eldst   <= ldst;
        end
    // EXE
    cla32 ret_addr (epc4,32'h4,1'b0,epc8); // PC+8
    assign sa = {eimm[5:0],eimm[31:6]}; // shift amount
    mux2x32 alu_ina (ea,sa,eshift,alua); // ALU input A
    mux2x32 alu_inb (eb,eimm,ealuimm,alub); // ALU input B
    mux2x32 save_pc8 (ealu0,epc8,ejal,ealu1); // PC+8 if jal
    mux2x32 read_cr0 (ealu1,c0reg,erc0,ealu); // read c0 regs
    wire z;
    alu al_unit (alua,alub,ealuc,ealu0,z); // ALU
    assign ern = ern0 | {5{ejal}}; // r31 for jal
    mux2x32 fwd_f_e (ed,e3d,efwdfe,eb); // forward fp result
    // EXE-MEM pipeline registers
    reg [31:0] pcm;
    reg        misbr,mldst;
    always @(negedge resetn or posedge clock)
        if (resetn == 0) begin
            mwfpr   <= 0;           mwreg   <= 0;
            mm2reg  <= 0;           mwmem   <= 0;
            malu    <= 0;           mb      <= 0;
            mrn     <= 0;           misbr   <= 0;
            pcm     <= 0;           mldst   <= 0;
        end else if (no_cache_stall) begin
            mwfpr   <= ewfpr & ~dtlb_exc; // cancel
            mwreg   <= ewreg & ~dtlb_exc; // cancel
            mwmem   <= ewmem & ~dtlb_exc; // cancel
            mldst   <= eldst & ~dtlb_exc; // cancel
            mm2reg  <= em2reg;
            malu    <= ealu;        mb      <= eb;
            mrn     <= ern;         misbr   <= eisbr;
```

```
        pcm     <= pce;
    end
// MEM
wire [19:0] dpattern = (dtlbwi | dtlbwr) ? enthi[19:0] : malu[31:12];
wire ma_unmapped     =  malu[31] & ~malu[30]; // 10x; malu: va of data
wire [31:0] m_addr   =  ma_unmapped? {1'b0,malu[30:0]} :
                                    {dpte_out[19:0],malu[11:0]};
wire  [2:0] drandom;
wire [23:0] dpte_out;
wire        dtlb_hit;
wire  [2:0] dvpn_index;
wire        dvpn_found;
tlb_8_entry dtlb (entlo[23:0], dtlbwi, dtlbwr, index[2:0], dpattern,
                  memclock, clock, resetn,
                  drandom, dpte_out, dtlb_hit, dvpn_index, dvpn_found);
wire dtlb_exc = ~dtlb_hit & ~ma_unmapped & mldst;
wire d_ready;
wire w_mem = mwmem & ~dtlb_exc; // cancel sw/swc1 in mem stage
d_cache dcache (m_addr,mb,mmo,mldst,w_mem,d_ready,clock,resetn,
                m_d_a,mem_data,mem_st_data,m_ld_st,m_st,m_d_ready);
// MEM-WB pipeline registers
reg [31:0] pcw;
reg        wisbr;
always @(negedge resetn or posedge clock)
    if (resetn == 0) begin
        wwfpr   <= 0;          wwreg   <= 0;
        wm2reg  <= 0;          wmo     <= 0;
        walu    <= 0;          wrn     <= 0;
        pcw     <= 0;          wisbr   <= 0;
    end else if (no_cache_stall) begin
        wwfpr   <= mwfpr & ~dtlb_exc; // cancel lwc1 in wb stage
        wwreg   <= mwreg & ~dtlb_exc; // cancel lw   in wb stage
        wm2reg  <= mm2reg;     wmo     <= mmo;
        walu    <= malu;       wrn     <= mrn;
        pcw     <= pcm;        wisbr   <= misbr;
    end
// WB
mux2x32 wb_sel (walu,wmo,wm2reg,wdi);
// for main_memory access
wire m_fetch,m_ld_st,m_st;
wire [31:0] m_i_a,m_d_a;
// mux, i_cache has higher priority than d_cache
wire        sel_i = i_cache_miss;
assign      mem_a = sel_i? m_i_a   : m_d_a;
assign mem_access = sel_i? m_fetch : m_ld_st;
assign mem_write  = sel_i? 1'b0    : m_st;
// demux
```

```
    wire   m_i_ready  = mem_ready &  sel_i;
    wire   m_d_ready  = mem_ready & ~sel_i;
    assign no_cache_stall = ~(~i_ready | mldst & ~d_ready);
endmodule
```

以下是控制部件模块 control。

```
module control (op,func,rs,rt,rd,fs,ft,rsrtequ,              // control unit
                ewfpr,ewreg,em2reg,ern,                      // from iu
                mwfpr,mwreg,mm2reg,mrn,                      // from iu
                e1w,e1n,e2w,e2n,e3w,e3n,stall_div_sqrt,st, // from fpu
                pcsource,wpcir,wreg,m2reg,wmem,jal,aluc,    // iu
                sta,aluimm,shift,sext,regrt,fwda,fwdb,      // iu
                swfp,fwdf,fwdfe,wfpr,                        // for lwc1,swc1
                fwdla,fwdlb,fwdfa,fwdfb,fc,wf,fasmds,       // used by fpu
                stall_lw,stall_fp,stall_lwc1,stall_swc1,    // for testing
                windex,wentlo,wcontx,wenthi,rc0,wc0,tlbwi,tlbwr, // for tlb
                c0rn,wepc,wcau,wsta,isbr,sepc,cancel,cause,exc,selpc,ldst,
                wisbr,ecancel,itlb_exc,dtlb_exc);
    input  rsrtequ, ewreg,em2reg,ewfpr, mwreg,mm2reg,mwfpr;
    input  e1w,e2w,e3w,stall_div_sqrt,st;
    input  [5:0] op,func;
    input  [4:0] rs,rt,rd,fs,ft,ern,mrn,e1n,e2n,e3n;
    input  [31:0] sta; // IM[7:0] : x,x,dtlb_exc,itlb_exc,ov,unimpl,sys,int
    output wpcir,wreg,m2reg,wmem,jal,aluimm,shift,sext,regrt;
    output swfp,fwdf,fwdfe;
    output fwdla,fwdlb,fwdfa,fwdfb;
    output wfpr,wf,fasmds;
    output [1:0] pcsource,fwda,fwdb;
    output [3:0] aluc;
    output [2:0] fc;
    output stall_lw,stall_fp,stall_lwc1,stall_swc1; // for testing
    output windex,wentlo,wcontx,wenthi,rc0,wc0,tlbwi,tlbwr; // for tlb
    output [1:0] c0rn,sepc,selpc;
    output wepc,wcau,wsta,isbr,cancel,exc,ldst;
    output [31:0] cause;
    input  wisbr,ecancel,itlb_exc,dtlb_exc;

    assign ldst = (i_lw | i_sw | i_lwc1 | i_swc1) & ~ecancel & ~dtlb_exc;
    assign isbr = i_beq | i_bne | i_j | i_jal;
    // itlb_exc dtlb_exc isbr wisbr EPC   sepc[1:0]
    // 1        x        0    x     V_PC  0 0
    // 1        x        1    x     PCD   0 1
    // 0        1        x    0     PCM   1 0
    // 0        1        x    1     PCW   1 1
    assign sepc[1] = ~itlb_exc & dtlb_exc;
    assign sepc[0] =  itlb_exc & isbr | ~itlb_exc & dtlb_exc & wisbr;
    assign exc     =  itlb_exc & sta[4] | dtlb_exc & sta[5]; // mask
```

```
assign cancel  =  exc;
// selpc
// 0 0 : npc
// 0 1 : epc
// 1 0 : EXC_BASE
// 1 1 : x
assign selpc[1] = exc;
assign selpc[0] = i_eret;
// op     rs    rt    rd          func
// 010000 00100 xxxxx xxxxx 00000 000000 mtc0 rt, rd; c0[rd] <-- gpr[rt]
// 010000 00000 xxxxx xxxxx 00000 000000 mfc0 rt, rd; gpr[rt] <-- c0[rd]
// 010000 10000 00000 00000 00000 000010 tlbwi
// 010000 10000 00000 00000 00000 000110 tlbwr
// 010000 10000 00000 00000 00000 011000 eret
wire   i_mtc0  = (op==6'h10) & (rs==5'h04) & (func==6'h00);
wire   i_mfc0  = (op==6'h10) & (rs==5'h00) & (func==6'h00);
wire   i_eret  = (op==6'h10) & (rs==5'h10) & (func==6'h18);
assign tlbwi   = (op==6'h10) & (rs==5'h10) & (func==6'h02);
assign tlbwr   = (op==6'h10) & (rs==5'h10) & (func==6'h06);
assign windex  = i_mtc0 & (rd==5'h00); // write index
assign wentlo  = i_mtc0 & (rd==5'h02); // write entry_lo
assign wcontx  = i_mtc0 & (rd==5'h04); // write context
assign wenthi  = i_mtc0 & (rd==5'h09); // write entry_hi
assign wsta    = i_mtc0 & (rd==5'h0c) | exc | i_eret; // write status
assign wcau    = i_mtc0 & (rd==5'h0d) | exc; // write cause
assign wepc    = i_mtc0 & (rd==5'h0e) | exc; // write epc
//wire rcontx  = i_mfc0 & (rd==5'h04); // read  context
wire rstatus   = i_mfc0 & (rd==5'h0c); // read  status
wire rcause    = i_mfc0 & (rd==5'h0d); // read  cause
wire repc      = i_mfc0 & (rd==5'h0e); // read  epc
assign c0rn[1] = rcause  | repc; // c0rn 00    01  10  11
assign c0rn[0] = rstatus | repc; //      contx sta cau epc
assign rc0     = i_mfc0; // read  c0 regs
assign wc0     = i_mtc0; // write c0 regs
wire [2:0] exccode;
// 100 00 itlb_exc
// 101 00 dtlb_exc
assign exccode[2] = itlb_exc | dtlb_exc;
assign exccode[1] = 1'b0;
assign exccode[0] = dtlb_exc;
assign cause      = {27'h0,exccode,2'b00};
wire r_type,i_add,i_sub,i_and,i_or,i_xor,i_sll,i_srl,i_sra,i_jr;
and(r_type,~op[5],~op[4],~op[3],~op[2],~op[1],~op[0]); // r format
and(i_add,r_type, func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_sub,r_type, func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_and,r_type, func[5],~func[4],~func[3], func[2],~func[1],~func[0]);
and(i_or, r_type, func[5],~func[4],~func[3], func[2],~func[1], func[0]);
```

```
and(i_xor,r_type, func[5],~func[4],~func[3], func[2], func[1],~func[0]);
and(i_sll,r_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_srl,r_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_sra,r_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_jr, r_type,~func[5],~func[4], func[3],~func[2],~func[1],~func[0]);
wire i_addi,i_andi,i_ori,i_xori,i_lw,i_sw,i_beq,i_bne,i_lui;
and(i_addi,~op[5],~op[4], op[3],~op[2],~op[1],~op[0]);
and(i_andi,~op[5],~op[4], op[3], op[2],~op[1],~op[0]);
and(i_ori, ~op[5],~op[4], op[3], op[2],~op[1], op[0]);
and(i_xori,~op[5],~op[4], op[3], op[2], op[1],~op[0]);
and(i_lw,   op[5],~op[4],~op[3],~op[2], op[1], op[0]);
and(i_sw,   op[5],~op[4], op[3],~op[2], op[1], op[0]);
and(i_beq, ~op[5],~op[4],~op[3], op[2],~op[1],~op[0]);
and(i_bne, ~op[5],~op[4],~op[3], op[2],~op[1], op[0]);
and(i_lui, ~op[5],~op[4], op[3], op[2], op[1], op[0]);
wire i_j,i_jal;
and(i_j,   ~op[5],~op[4],~op[3],~op[2], op[1],~op[0]);
and(i_jal, ~op[5],~op[4],~op[3],~op[2], op[1], op[0]);
wire f_type,i_lwc1,i_swc1,i_fadd,i_fsub,i_fmul,i_fdiv,i_fsqrt;
and(f_type,~op[5], op[4],~op[3],~op[2],~op[1], op[0]); // f format
and(i_lwc1, op[5], op[4],~op[3],~op[2],~op[1], op[0]);
and(i_swc1, op[5], op[4], op[3],~op[2],~op[1], op[0]);
and(i_fadd,f_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_fsub,f_type,~func[5],~func[4],~func[3],~func[2],~func[1], func[0]);
and(i_fmul,f_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_fdiv,f_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_fsqrt,f_type,~func[5],~func[4],~func[3],func[2],~func[1],~func[0]);
wire i_rs = i_add  | i_sub | i_and  | i_or  | i_xor | i_jr  | i_addi |
            i_andi | i_ori | i_xori | i_lw  | i_sw  | i_beq | i_bne  |
            i_lwc1 | i_swc1;
wire i_rt = i_add  | i_sub | i_and  | i_or  | i_xor | i_sll | i_srl  |
            i_sra  | i_sw  | i_beq  | i_bne | i_mtc0;
assign stall_lw = ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) |
                                                 i_rt & (ern == rt));
reg [1:0] fwda, fwdb;
always @ (ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or
          rt) begin
    fwda = 2'b00; // default forward a: no hazards
    if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
        fwda = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
            fwda = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                fwda = 2'b11; // select mem_lw
            end
```

```
            end
        end
        fwdb = 2'b00; // default forward b: no hazards
        if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
            fwdb = 2'b01; // select exe_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
                fwdb = 2'b10; // select mem_alu
            end else begin
                if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                    fwdb = 2'b11; // select mem_lw
                end
            end
        end
    end
    assign wreg    =(i_add  | i_sub  | i_and  | i_or   | i_xor | i_sll  |
                     i_srl  | i_sra  | i_addi | i_andi | i_ori | i_xori |
                     i_lw   | i_lui  | i_jal  | i_mfc0) &
                     wpcir  & ~ecancel & ~dtlb_exc;
    assign regrt  = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui |
                    i_lwc1 | i_mfc0;
    assign jal    = i_jal;
    assign m2reg  = i_lw;
    assign shift  = i_sll  | i_srl  | i_sra;
    assign aluimm = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui |
                    i_sw | i_lwc1 | i_swc1;
    assign sext   = i_addi | i_lw | i_sw | i_beq | i_bne | i_lwc1 | i_swc1;
    assign aluc[3] = i_sra;
    assign aluc[2] = i_sub | i_or  | i_srl | i_sra | i_ori  | i_lui;
    assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_beq  |
                     i_bne | i_lui;
    assign aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori;
    assign wmem    = (i_sw | i_swc1) & wpcir & ~ecancel & ~dtlb_exc;
    assign pcsource[1] = i_jr | i_j | i_jal;
    assign pcsource[0] = i_beq & rsrtequ | i_bne & ~rsrtequ | i_j | i_jal;
    // fop:  000: fadd  001: fsub  01x: fmul  10x: fdiv  11x: fsqrt
    wire [2:0]  fop;
    assign fop[0] = i_fsub; // fpu operation control code
    assign fop[1] = i_fmul | i_fsqrt;
    assign fop[2] = i_fdiv | i_fsqrt;
    // stall caused by fp data harzards
    wire i_fs = i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt; // use fs
    wire i_ft = i_fadd | i_fsub | i_fmul | i_fdiv;            // use ft
    assign stall_fp = (e1w & (i_fs & (e1n == fs) | i_ft & (e1n == ft))) |
                      (e2w & (i_fs & (e2n == fs) | i_ft & (e2n == ft)));
    assign fwdfa = e3w & (e3n == fs); // forward fpu e3d to fp a
    assign fwdfb = e3w & (e3n == ft); // forward fpu e3d to fp b
```

```
    assign wfpr  = i_lwc1 & wpcir & ~ecancel & ~dtlb_exc; // fp regfile we
    assign fwdla = mwfpr & (mrn == fs); // forward mmo to fp a
    assign fwdlb = mwfpr & (mrn == ft); // forward mmo to fp b
    assign stall_lwc1 = ewfpr & (i_fs & (ern == fs) | i_ft & (ern == ft));
    assign swfp  = i_swc1; // select signal
    assign fwdf  = swfp & e3w & (ft == e3n); // forward to id  stage
    assign fwdfe = swfp & e2w & (ft == e2n); // forward to exe stage
    assign stall_swc1 = swfp & e1w & (ft == e1n); // stall
    assign wpcir = ~(stall_div_sqrt | stall_others);
    wire stall_others = stall_lw | stall_fp | stall_lwc1 | stall_swc1 | st;
    assign fc = fop & {3{~stall_others}};
    assign wf = i_fs & wpcir;
    assign fasmds = i_fs;
endmodule
```

以下是主存模块，分为 4 个区间。

```
module physical_memory #(parameter A_WIDTH = 32)
    (a, dout, din, strobe, rw, ready, clk, memclk, clrn);
    input [A_WIDTH-1:0]  a;
    output [31:0]        dout;
    input [31:0]         din;
    input                strobe;
    input                rw;
    output               ready;
    input                clk, memclk, clrn;
    // for memory ready
    reg [2:0]            wait_counter;
    reg                  ready;
    always @ (negedge clrn or posedge clk) begin
        if (clrn == 0) begin
            wait_counter <= 3'b0;
        end else begin
            if (strobe) begin
                if (wait_counter == 3'h5) begin
                    ready <= 1'b1;
                    wait_counter <= 3'b0;
                end else begin
                    ready <= 1'b0;
                    wait_counter <= wait_counter + 3'b1;
                end
            end else begin
                ready <= 1'b0;
                wait_counter <= 3'b0;
            end
        end
    end
```

```
// 31 30 29 28 ... 15 14 13 12 ... 3  2  1  0
//  0  0  0  0      0  0  0  0      0  0  0  0 (0) 0x0000_0000
//  0  0  0  1      0  0  0  0      0  0  0  0 (1) 0x1000_0000
//  0  0  1  0      0  0  0  0      0  0  0  0 (2) 0x2000_0000
//  0  0  1  0      0  0  1  0      0  0  0  0 (3) 0x2000_2000
wire [31:0] m_out32 = a[13] ? mem_data_out3 : mem_data_out2;
wire [31:0] m_out10 = a[28] ? mem_data_out1 : mem_data_out0;
wire [31:0] mem_out = a[29] ? m_out32       : m_out10;
assign      dout    = ready ? mem_out       : 32'hzzzz_zzzz;

// (0) 0x0000_0000- (virtual address 0x8000_0000-)
wire [31:0] mem_data_out0;
wire        write_enable0 = ~a[29] & ~a[28] & rw & ~clk;
lpm_ram_dq ram0 (.data(din),.address(a[8:2]),
                 .we(write_enable0),.inclock(memclk),
                 .outclock(memclk&strobe),.q(mem_data_out0));
defparam   ram0.lpm_width    = 32;
defparam   ram0.lpm_widthad =  7;
defparam   ram0.lpm_indata   = "registered";
defparam   ram0.lpm_outdata = "registered";
defparam   ram0.lpm_file     = "cpu_cache_tlb_0.mif";
defparam   ram0.lpm_address_control = "registered";

// (1) 0x1000_0000- (virtual address 0x9000_0000-)
wire [31:0] mem_data_out1;
wire        write_enable1 = ~a[29] &  a[28] & rw & ~clk;
lpm_ram_dq ram1 (.data(din),.address(a[8:2]),
                 .we(write_enable1),.inclock(memclk),
                 .outclock(memclk&strobe),.q(mem_data_out1));
defparam   ram1.lpm_width    = 32;
defparam   ram1.lpm_widthad =  7;
defparam   ram1.lpm_indata   = "registered";
defparam   ram1.lpm_outdata = "registered";
defparam   ram1.lpm_file     = "cpu_cache_tlb_1.mif";
defparam   ram1.lpm_address_control = "registered";

// (2) 0x2000_0000- (mapped va 0x0000_0000-)
wire [31:0] mem_data_out2;
wire        write_enable2 = a[29] & ~a[13] & rw & ~clk;
lpm_ram_dq ram2 (.data(din),.address(a[8:2]),
                 .we(write_enable2),.inclock(memclk),
                 .outclock(memclk&strobe),.q(mem_data_out2));
defparam   ram2.lpm_width    = 32;
defparam   ram2.lpm_widthad =  7;
defparam   ram2.lpm_indata   = "registered";
```

```
    defparam   ram2.lpm_outdata = "registered";
    defparam   ram2.lpm_file     = "cpu_cache_tlb_2.mif";
    defparam   ram2.lpm_address_control = "registered";

    // (3) 0x2000_2000- (mapped va 0x0000_0000-)
    wire [31:0] mem_data_out3;
    wire        write_enable3 = a[29] & a[13] & rw & ~clk;
    lpm_ram_dq ram3 (.data(din),.address(a[8:2]),
                     .we(write_enable3),.inclock(memclk),
                     .outclock(memclk&strobe),.q(mem_data_out3));
    defparam   ram3.lpm_width    = 32;
    defparam   ram3.lpm_widthad = 7;
    defparam   ram3.lpm_indata   = "registered";
    defparam   ram3.lpm_outdata = "registered";
    defparam   ram3.lpm_file     = "cpu_cache_tlb_3.mif";
    defparam   ram3.lpm_address_control = "registered";
endmodule
```

13.5 带有 Cache 及 TLB 的 CPU 的测试程序和仿真波形

本节给出 CPU 的测试程序及数据。以下是初始化和异常处理程序。

```
DEPTH = 128;          % Memory depth and width are required   %
WIDTH = 32;           % Enter a decimal number                %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless   %
                      % otherwise specified, radixes = HEX    %
CONTENT
BEGIN
   % physical address = 0x0000_0000                                      %
   % reset entry,  va = 0x8000_0000                                      %
 0: 08000070; % (00) j    init              # jump to init               %
 1: 00000000; % (04) nop                                                 %
   % EXC_BASE:                              # exception/interrupt entry  %
 2: 401a6800; % (08) mfc0 r26, C0_CAUSE     # read cp0 Cause reg         %
 3: 335a001c; % (0c) andi r26, r26, 0x1c    # get ExcCode, 3 bits here   %
 4: 3c1b8000; % (10) lui  r27, 0x8000       #                            %
 5: 037ad825; % (14) or   r27, r27, r26     #                            %
 6: 8f7b0040; % (18) lw   r27, j_table(r27)# get address from table     %
 7: 00000000; % (1c) nop                    #                            %
 8: 03600008; % (20) jr   r27               # jump to that address       %
 9: 00000000; % (24) nop                    #                            %
[a..f] : 0;
   % j_table:              # address table for exception and interrupt   %
10: 80000030; % (40)  int_entry  # 0. address for interrupt              %
11: 8000003c; % (44)  sys_entry  # 1. address for Syscall                %
```

```
12: 80000054; % (48)  uni_entry  # 2. address for Unimpl. inst.        %
13: 80000068; % (4c)  ovf_entry  # 3. address for Overflow             %
14: 800000c0; % (50)  itlb_entry # 4. address for itlb miss            %
15: 80000140; % (54)  dtlb_entry # 5. address for dtlb miss            %
16: 80000000; % (58)                                                   %
17: 80000000; % (5c)                                                   %
[18..2f] : 0;
    % itlb_entry:                                                      %
30: 3c1b8000; % (c0)  lui  r27, 0x8000       # 0x800001f8: counter     %
31: 8f7a01f8; % (c4)  lw   r26, 0x1f8(r27)   # load itlb index counter %
32: 235a0001; % (c8)  addi r26, r26, 1       # index + 1               %
33: 335a0007; % (cc)  andi r26, r26, 7       # 3-bit index             %
34: af7a01fc; % (d0)  sw   r26, 0x1fc(r27)   # store index             %
35: 3c1b0000; % (d4)  lui  r27, 0x0000       # itlb tag                %
36: 037ad025; % (d8)  or   r26, r27, r26     # itlb tag and index      %
37: 409a0000; % (dc)  mtc0 r26, C0_INDEX     # move to c0 index        %
38: 401b2000; % (e0)  mfc0 r27, C0_CONTEXT   # move from c0 context    %
39: 8f7a0000; % (e4)  lw   r26, 0x0(r27)     # get pte                 %
3a: 409a1000; % (e8)  mtc0 r26, C0_ENTRY_LO  # move to c0 entry_lo     %
3b: 001bd280; % (ec)  sll  r26, r27, 10      # get bad vpn             %
3c: 001ad302; % (f0)  srl  r26, r26, 12      # for c0 entry_hi         %
3d: 409a4800; % (f4)  mtc0 r26, C0_ENTRY_HI  # move to entry_hi        %
3e: 42000002; % (f8)  tlbwi                  # update itlb             %
3f: 42000018; % (fc)  eret                   # return from exception   %
40: 00000000; % (100) nop                    #                         %
[41..4f] : 0;
    % dtlb_entry:                                                      %
50: 3c1b8000; % (140) lui  r27, 0x8000       # 0x800001fc: counter     %
51: 8f7a01fc; % (144) lw   r26, 0x1fc(r27)   # load dtlb index counter %
52: 235a0001; % (148) addi r26, r26, 1       # index + 1               %
53: 335a0007; % (14c) andi r26, r26, 7       # 3-bit index             %
54: af7a01fc; % (150) sw   r26, 0x1fc(r27)   # store index             %
55: 3c1b4000; % (154) lui  r27, 0x4000       # dtlb tag                %
56: 037ad025; % (158) or   r26, r27, r26     # dtlb tag and index      %
57: 409a0000; % (15c) mtc0 r26, C0_INDEX     # move to c0 index        %
58: 401b2000; % (160) mfc0 r27, C0_CONTEXT   # move from c0 context    %
59: 8f7a0000; % (164) lw   r26, 0x0(r27)     # get pte                 %
5a: 409a1000; % (168) mtc0 r26, C0_ENTRY_LO  # move to c0 entry_lo     %
5b: 001bd280; % (16c) sll  r26, r27, 10      # get bad vpn             %
5c: 001ad302; % (170) srl  r26, r26, 12      # for c0 entry_hi         %
5d: 409a4800; % (174) mtc0 r26, C0_ENTRY_HI  # move to entry_hi        %
5e: 42000002; % (178) tlbwi                  # update dtlb             %
5f: 42000018; % (17c) eret                   # return from exception   %
60: 00000000; % (180) nop                    #                         %
[61..6f] : 0;

    % init                                                             %
```

```
70: 40800000; % (1c0) mtc0 r0,  C0_INDEX    # C0_INDEX <-- 0 (itlb[0])  %
71: 3c1b9000; % (1c4) lui  r27, 0x9000      # page table base           %
72: 8f7a0000; % (1c8) lw   r26, 0x0(r27)    # 1st entry of page table   %
73: 409a1000; % (1cc) mtc0 r26, C0_ENTRY_LO # C0_ENTRY_LO <-- v,d,c,pfn %
74: 3c1a0000; % (1d0) lui  r26, 0x0         # va (=0) for C0_ENTRY_HI   %
75: 409a4800; % (1d4) mtc0 r26, C0_ENTRY_HI # C0_ENTRY_HI <-- vpn (0)   %
76: 42000002; % (1d8) tlbwi                 # write itlb for user prog  %
77: 409b2000; % (1dc) mtc0 r27, C0_CONTEXT  # C0_CONTEXT <-- PTEBase    %
78: 341a003f; % (1e0) ori  r26, r0, 0x3f    # enable exceptions         %
79: 409a6000; % (1e4) mtc0 r26, C0_STATUS   # C0_STATUS <-- 0..00111111 %
7a: 3c010000; % (1e8) lui  r1, 0x0          # va = 0x0000_0000          %
7b: 00200008; % (1ec) jr   r1               # jump to user program      %
7c: 00000000; % (1f0) nop                   #                           %
7d: 00000000; % (1f4) nop                   #                           %
7e: 00000000; % (1f8) .data 0               # itlb index counter        %
7f: 00000000; % (1fc) .data 0               # dtlb index counter        %
END ;
```

页表，此处很小，实际很大。系统软件可以维护这个页表：

```
DEPTH = 128;          % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
    % physical address = 0x1000_0000 %
    % page table,    va = 0x9000_0000 %
 0: 00820000; % (00) va: 0000_0000 --> pa: 20000000 ; 1 of 8: valid bit %
 1: 00820002; % (04) va: 0000_1000 --> pa: 20002000 ; 1 of 8: valid bit %
 2: 00820001; % (08) va: 0000_2000 --> pa: 20001000 ; 1 of 8: valid bit %
 3: 008200f0; % (0c) va: 0000_3000 --> pa: 200f0000 ; 1 of 8: valid bit %
[4..7F] : 00000000;
END ;
```

IU / FPU 测试程序：

```
DEPTH = 128;          % Memory depth and width are required    %
WIDTH = 32;           % Enter a decimal number                 %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless    %
                      % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
  % physical address = 0x2000_0000 %
 0 : 20011100; %(20000000)   addi  r1,r0,0x1100  # address of data[0]   %
 1 : C4200000; %(20000004)   lwc1  f0,  0x0(r1)  # load fp data         %
```

```
 2 : C4210050; %(20000008)   lwc1  f1, 0x50(r1)  # load fp data           %
 3 : C4220054; %(2000000C)   lwc1  f2, 0x54(r1)  # load fp data           %
 4 : C4230058; %(20000010)   lwc1  f3, 0x58(r1)  # load fp data           %
 5 : C424005C; %(20000014)   lwc1  f4, 0x5c(r1)  # load fp data           %
 6 : 46002100; %(20000018)   add.s f4,  f4, f0   # f4: stall 1            %
 7 : 460418C1; %(2000001C)   sub.s f3,  f3, f4   # f4: stall 2            %
 8 : 46022082; %(20000020)   mul.s f2,  f4, f2   # mul                    %
 9 : 46040842; %(20000024)   mul.s f1,  f1, f4   # mul                    %
 A : E4210070; %(20000028)   swc1  f1, 0x70(r1)  # f1: stall 1            %
 B : E4220074; %(2000002C)   swc1  f2, 0x74(r1)  # store fp data          %
 C : E4230078; %(20000030)   swc1  f3, 0x78(r1)  # store fp data          %
 D : E424007C; %(20000034)   swc1  f4, 0x7c(r1)  # store fp data          %
 E : 20020004; %(20000038)   addi  r2,  r0,  4   # counter                %
 F : C4230000; %(2000003C) l3: lwc1  f3, 0x0(r1) # load fp data           %
10 : C4210050; %(20000040)   lwc1  f1,  0x50(r1) # load fp data           %
11 : 46030840; %(20000044)   add.s f1,  f1, f3   # stall 1                %
12 : 46030841; %(20000048)   sub.s f1,  f1, f3   # stall 2                %
13 : E4210030; %(2000004C)   swc1  f1,  0x30(r1) # stall 1                %
14 : C4051104; %(20000050)   lwc1  f5,0x1104(r0) # load fp data           %
15 : C4061108; %(20000054)   lwc1  f6,0x1108(r0) # load fp data           %
16 : C408110C; %(20000058)   lwc1  f8,0x110c(r0) # load fp data           %
17 : 460629C3; %(2000005C)   div.s f7,  f5, f6   # div                    %
18 : 46004244; %(20000060)   sqrt.s f9,  f8      # sqrt                   %
19 : 46004A84; %(20000064)   sqrt.s f10, f9      # sqrt                   %
1A : 2042FFFF; %(20000068)   addi  r2,  r2, -1   # counter - 1            %
1B : 1440FFF3; %(2000006C)   bne   r2,  r0, l3   # finish?                %
1C : 20210004; %(20000070)   addi  r1,  r1,  4   # address+4, DELAY SLOT  %
1D : 3c010000; %(20000074) iu_test:lui  r1, 0    # address of data[0]     %
1E : 34241150; %(20000078)   ori  r4, r1, 0x1150 # address of data[0]     %
1F : 0c000038; %(2000007C) call:   jal  sum      # call function          %
20 : 20050004; %(20000080) dslot1: addi r5,r0,4  # DELYED SLOT(DS)        %
21 : ac820000; %(20000084) return: sw  r2, 0(r4) # store result           %
22 : 8c890000; %(20000088)   lw   r9, 0(r4)      # check sw               %
23 : 01244022; %(2000008C)   sub  r8, r9, r4     # sub: r8 <-- r9 - r4    %
24 : 20050003; %(20000090)   addi r5, r0,  3     # counter                %
25 : 20a5ffff; %(20000094) loop2: addi r5,r5,-1  # counter - 1            %
26 : 34a8ffff; %(20000098)   ori  r8, r5, 0xffff # zero-extend: 0000ffff  %
27 : 39085555; %(2000009C)   xori r8, r8, 0x5555 # zero-extend: 0000aaaa  %
28 : 2009ffff; %(200000A0)   addi r9, r0, -1     # sign-extend: ffffffff  %
29 : 312affff; %(200000A4)   andi r10, r9,0xffff # zero-extend: 0000ffff  %
2A : 01493025; %(200000A8)   or   r6, r10, r9    # or:  ffffffff          %
2B : 01494026; %(200000AC)   xor  r8, r10, r9    # xor: ffff0000          %
2C : 01463824; %(200000B0)   and  r7, r10, r6    # and: 0000ffff          %
2D : 10a00003; %(200000B4)   beq  r5, r0, shift  # if r5 = 0, goto shift  %
2E : 00000000; %(200000B8) dslot2: nop           # DS                     %
2F : 08000025; %(200000BC)   j    loop2          # jump loop2             %
30 : 00000000; %(200000C0) dslot3: nop           # DS                     %
```

```
31 : 2005ffff; %(200000C4) shift:  addi r5,r0,-1 # r5   = ffffffff      %
32 : 000543c0; %(200000C8)   sll  r8, r5, 15     # <<  15 = ffff8000     %
33 : 00084400; %(200000CC)   sll  r8, r8, 16     # <<  16 = 80000000     %
34 : 00084403; %(200000D0)   sra  r8, r8, 16     # >>> 16 = ffff8000     %
35 : 00084c32; %(200000D4)   srl  r8, r8, 15     # >>  15 = 0001ffff     %
36 : 08000036; %(200000D8) finish: j finish      # dead loop             %
37 : 00000000; %(200000DC) dslot4: nop           # DS                    %
38 : 00004020; %(200000E0) sum: add  r8, r0, r0  # sum                   %
39 : 8c890000; %(200000E4) loop: lw r9, 0(r4)    # load data             %
3A : 01094020; %(200000E8)   add  r8, r8, r9     # sum                   %
3B : 20a5ffff; %(200000EC)   addi r5, r5, -1     # counter - 1           %
3C : 14a0fffc; %(200000F0)   bne  r5, r0, loop   # finish?               %
3D : 20840004; %(200000F4) dslot5: addi r4,r4,4  # address + 4, DS       %
3E : 03e00008; %(200000F8)   jr   r31            # return                %
3F : 00081000; %(200000FC) dslot6: sll  r2,r8,0  # move res. to v0, DS   %
[40..7F] : 0;
END ;
```

IU/FPU 测试数据：

```
DEPTH = 128;           % Memory depth and width are required    %
WIDTH = 32;            % Enter a decimal number                 %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless    %
                       % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
    % physical address = 0x2000_2000 %
[0..3F] :   0; % (20002000..200020FC) 0 %
40 : BF800000; % (20002100) 1 01111111 00..0 fp -1 %
41 : 40800000; % (20002104) %
42 : 40000000; % (20002108) %
43 : 41100000; % (2000210C) %
[44..53] :  0; % (20002110..2000214C) 0 %
54 : 40C00000; % (20002150) 0 10000001 10..0 data[0] 4.5%
55 : 41C00000; % (20002154) 0 10000011 10..0 data[1] %
56 : 43C00000; % (20002158) 0 10000111 10..0 data[2] %
57 : 47C00000; % (2000215C) 0 10001111 10..0 data[3] %
[58..7F] :  0; % (20002160..200021FC) 0 %
END ;
```

图 13.10 ~ 图 13.12 是执行测试程序时的部分波形。复位后，CPU 从虚拟地址 0x80000000 开始执行程序，主要工作是为用户程序初始化一个 ITLB 项。然后转去执行用户程序。当用户程序引起 DTLB 不命中异常时，转去执行异常处理程序，填充 DTLB 项，然后返回。为了看波形图方便，我们假设 Cache 不命中时需要 6 个周期访问存储器 (实际不止 6 个周期)。

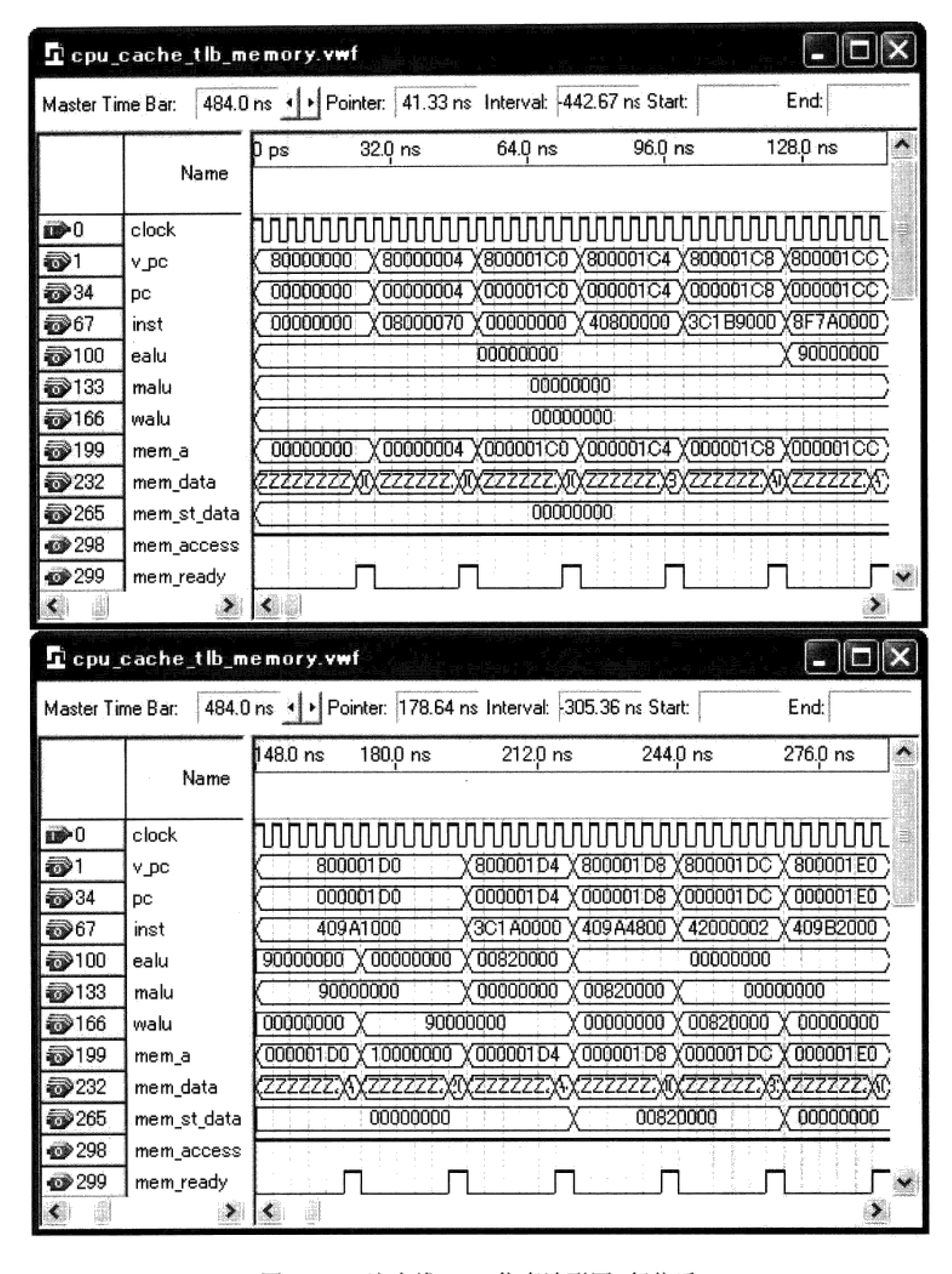


图 13.10　流水线 CPU 仿真波形图 (复位后)

图 13.10 中的 v_pc 是指令的虚拟地址，pc 是实际地址。这个区域的地址转换不经过 ITLB。由于是刚开始，指令 Cache 一定是不命中。

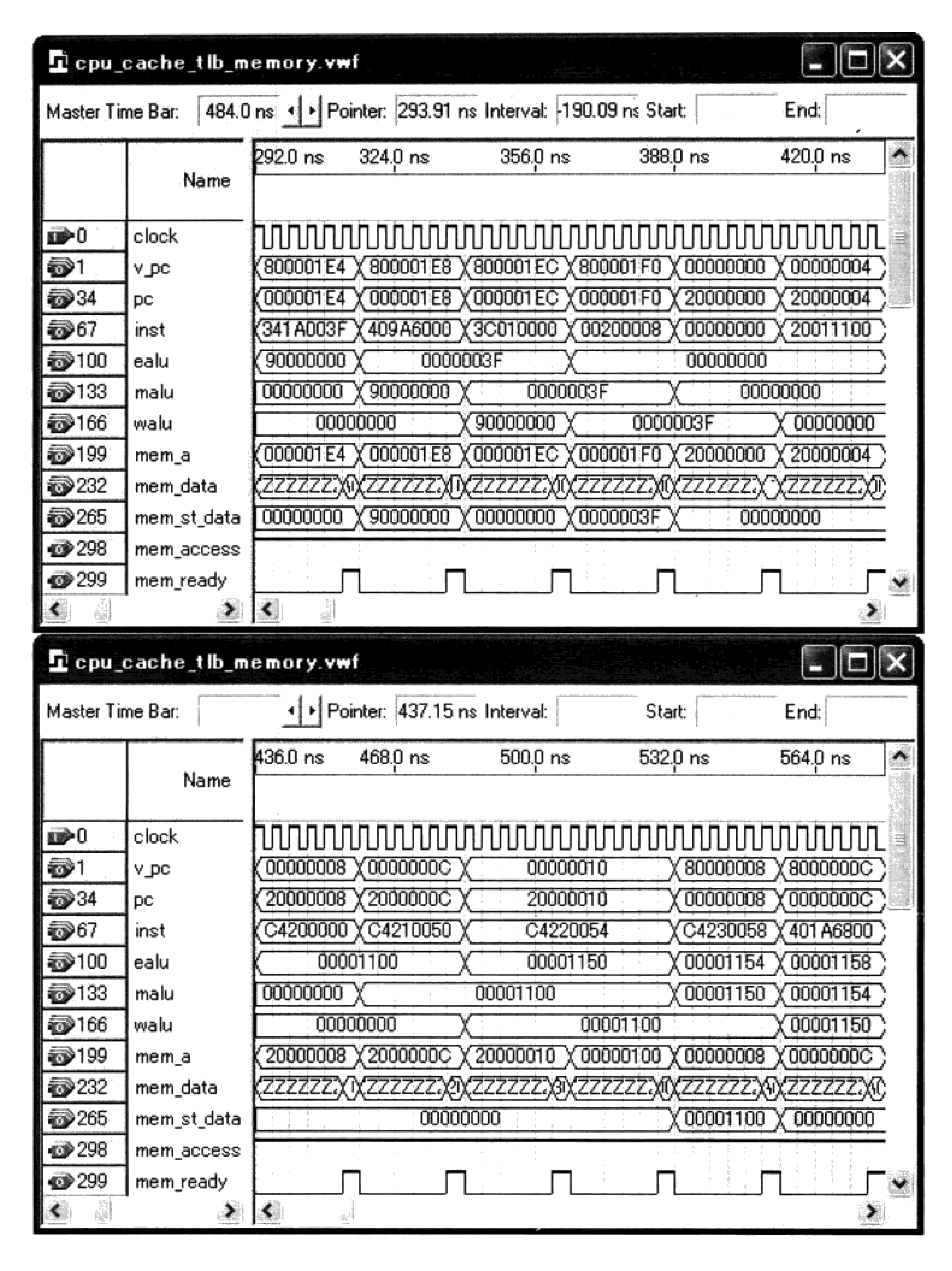


图 13.11 流水线 CPU 仿真波形图 (转用户程序)

图 13.11 上半部分示出的是初始化完成后转去执行用户程序的波形。下半部分示出的是 lwc1 指令执行到 MEM 级时 DTLB 不命中而转去执行异常处理程序的波形。

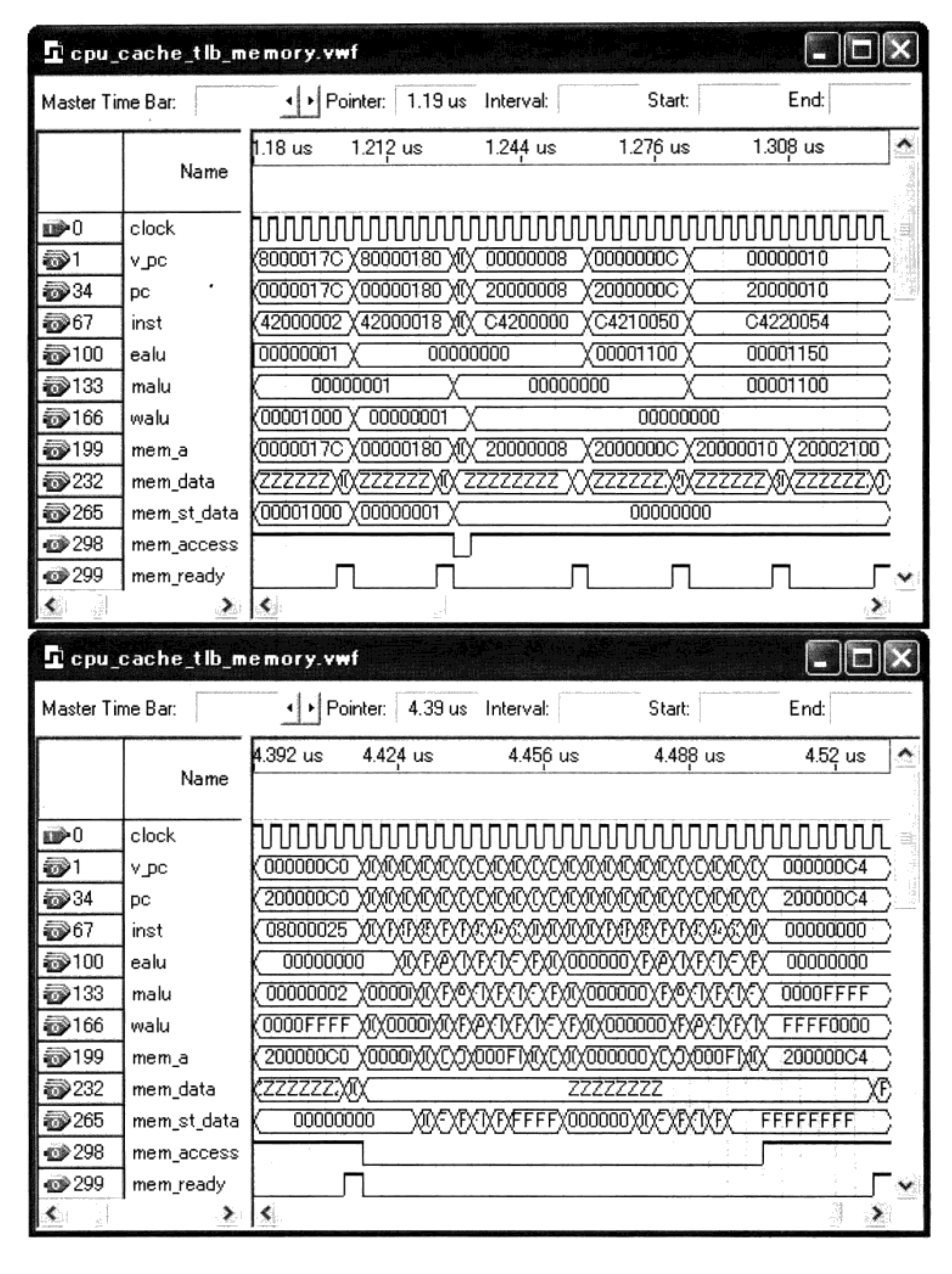


图 13.12　流水线 CPU 仿真波形图 (从异常返回和指令 Cache 命中)

图 13.12 上半部分是从异常处理返回时的波形。返回地址是 0x00000004，即重新执行 lwc1 指令 (Cache 命中)。下半部分是指令 Cache 命中时的波形。

13.6 习题

1. 重新设计 CPU，使其能够处理外部中断和异常，包括系统调用、未实现的指令、带符号数结果溢出以及浮点结果异常等。
2. 重新设计 CPU，使用 EntryLo0 和 EntryLo1。
3. 如何用软件实现各种 TLB 的替换策略?
4. 操作系统是如何管理页表的?

第 14 章　多核 CPU 及其 Verilog HDL 设计

本章介绍多核 CPU 的概念、结构、设计方法及 Cache 一致性问题，并给出一个双核 CPU 的具体设计的 Verilog HDL 代码及仿真波形。

14.1　多核 CPU 概述

14.1.1　多核 CPU 的基本概念

随着半导体工艺技术的提高，在一个芯片内能集成越来越多的晶体管，并且 CPU 的时钟频率也越来越快。但时钟频率不可能无限制地提高。我们知道，光在真空中的传播速度是 299 792 458m/s。电子信号的传播速度大约是光速的 2/3，大约是 200 000 000m/s。如果时钟频率是 4GHz，则在一个时钟周期内，电子信号仅能前进 5 cm。因此信号的延迟也是不得不考虑的问题。

如何充分有效地使用这些晶体管是 CPU 设计者面临的一个挑战。虽然超标量 (Superscalar) 和超长指令字 (Very Long Instruction Word，VLIW) 技术在不提高时钟频率的情况下能改善 CPU 的性能，但由于指令之间的相关性，性能的改进并不十分明显。多线程 (Multithreading) 技术也是改进 CPU 性能的有效方法，但由于复杂的执行部件被所有线程所共享，增加了电路的设计复杂性，线程数不可能很多。

多核 (Multi-Core) 技术至少在目前的 CPU 设计中被广泛地使用。它的中心思想是在一个芯片内设计多个“核” (Core)。图 14.1 示出了从单核到 16 核 CPU 的示意图。如果每个核具有相同的电路，则多核 CPU 的设计要比多线程简单。缺点是执行部件的利用率没有多线程高。

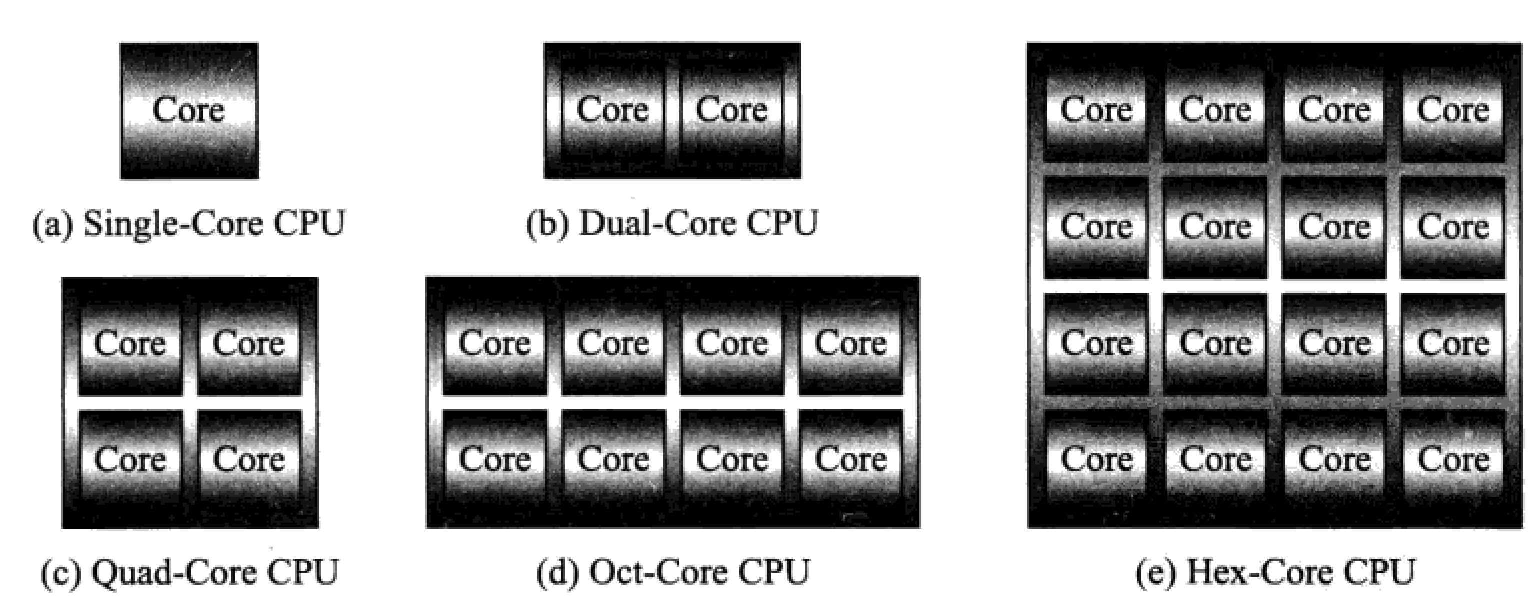
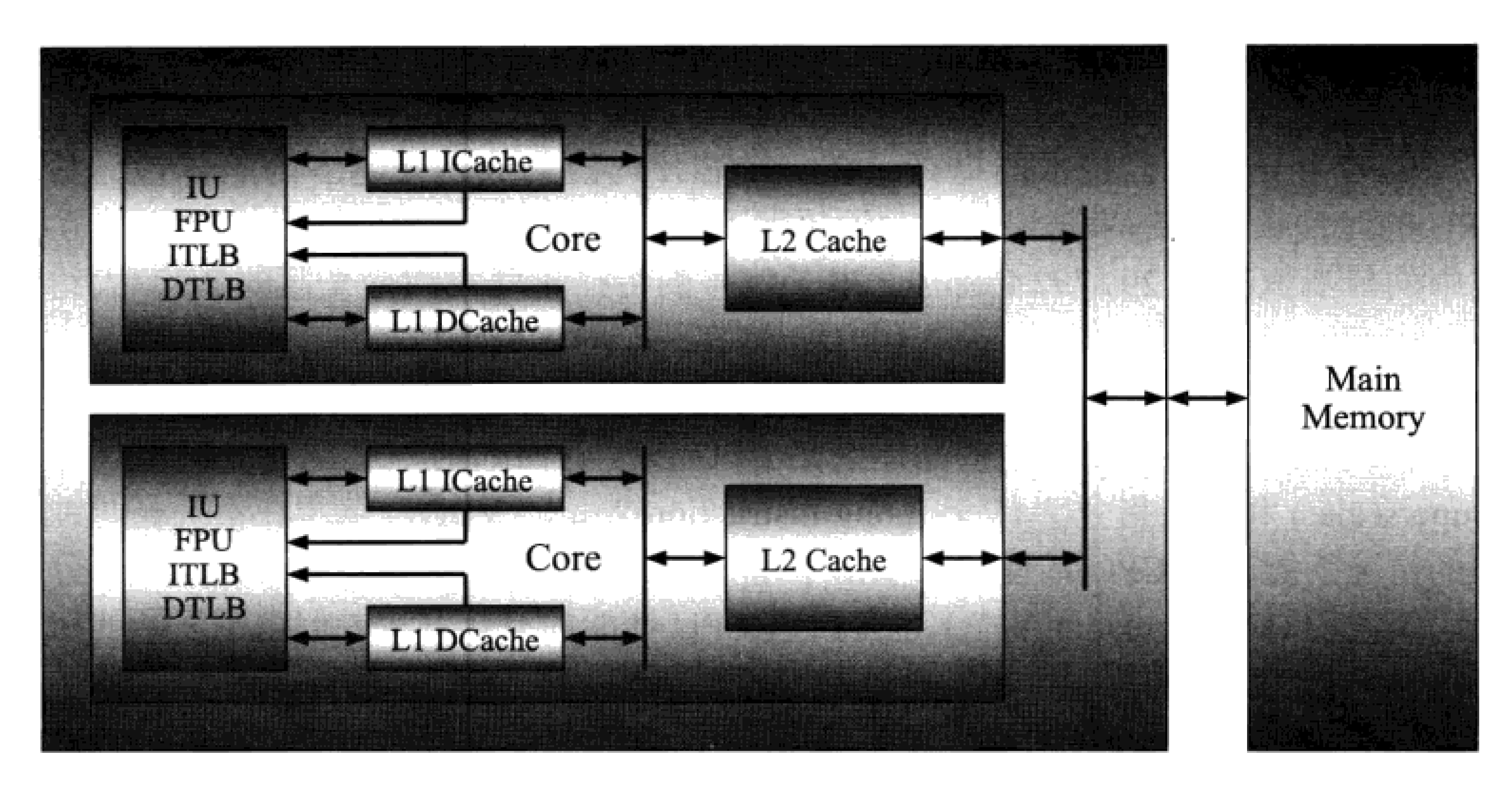


图 14.1　多核 CPU

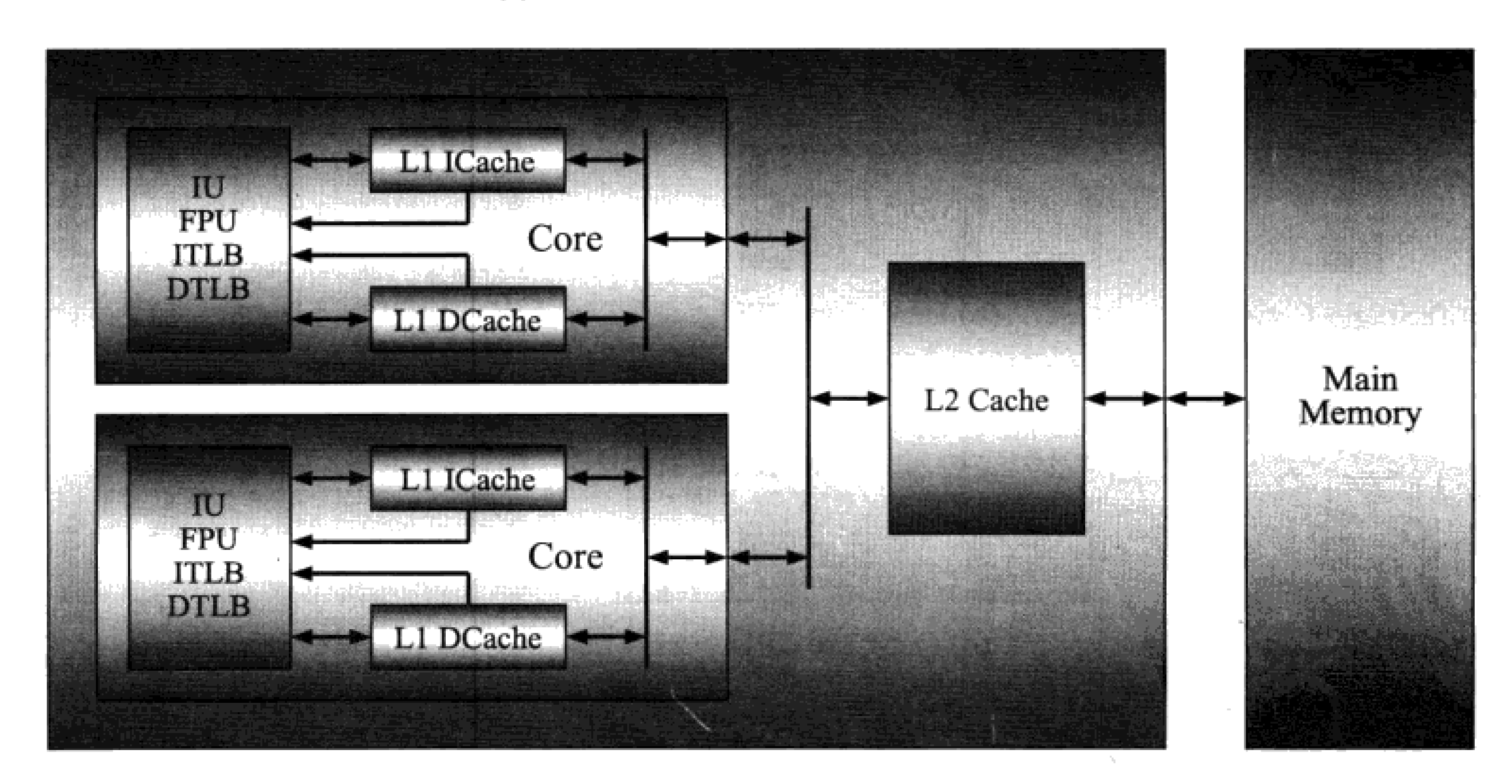
多核的名称由工业界提出。说穿了，一个核就是一个 CPU，多核就是在芯片的级别上实现“多处理机” (Multiprocessors) 技术。比多核更早出现的名称是学术界提

出的“Chip-Multiprocessors”。工业界有时不愿原封不动地使用学术界提出的名称。骨子里二者相同，然而你不得不承认 Multi-Core 读起来更简洁些。

如果核的数量不是很多，一般都采用共享总线的结构。图 14.2 示出了多核 CPU 的两种常用的结构。两种结构中每个核都有各自的第一级指令 Cache 和数据 Cache。图 14.2(a) 是一种比较简单的结构，每个核都有各自的第二级 Cache；图 14.2(b) 是整个 CPU 只有一个第二级 Cache，由所有核共享。不管是哪种结构，与外部的总线接口一般都只有一个。



(a) 多核 CPU: 独立的第二级 Cache



(b) 多核 CPU: 共享的第二级 Cache

图 14.2　两种不同结构的共享总线多核 CPU

如果使用第三级 Cache，则第二级 Cache 往往为每个核专有。如果核的数量很多，往往需要互联开关把所有的核连接在一起，如图 14.3 所示。

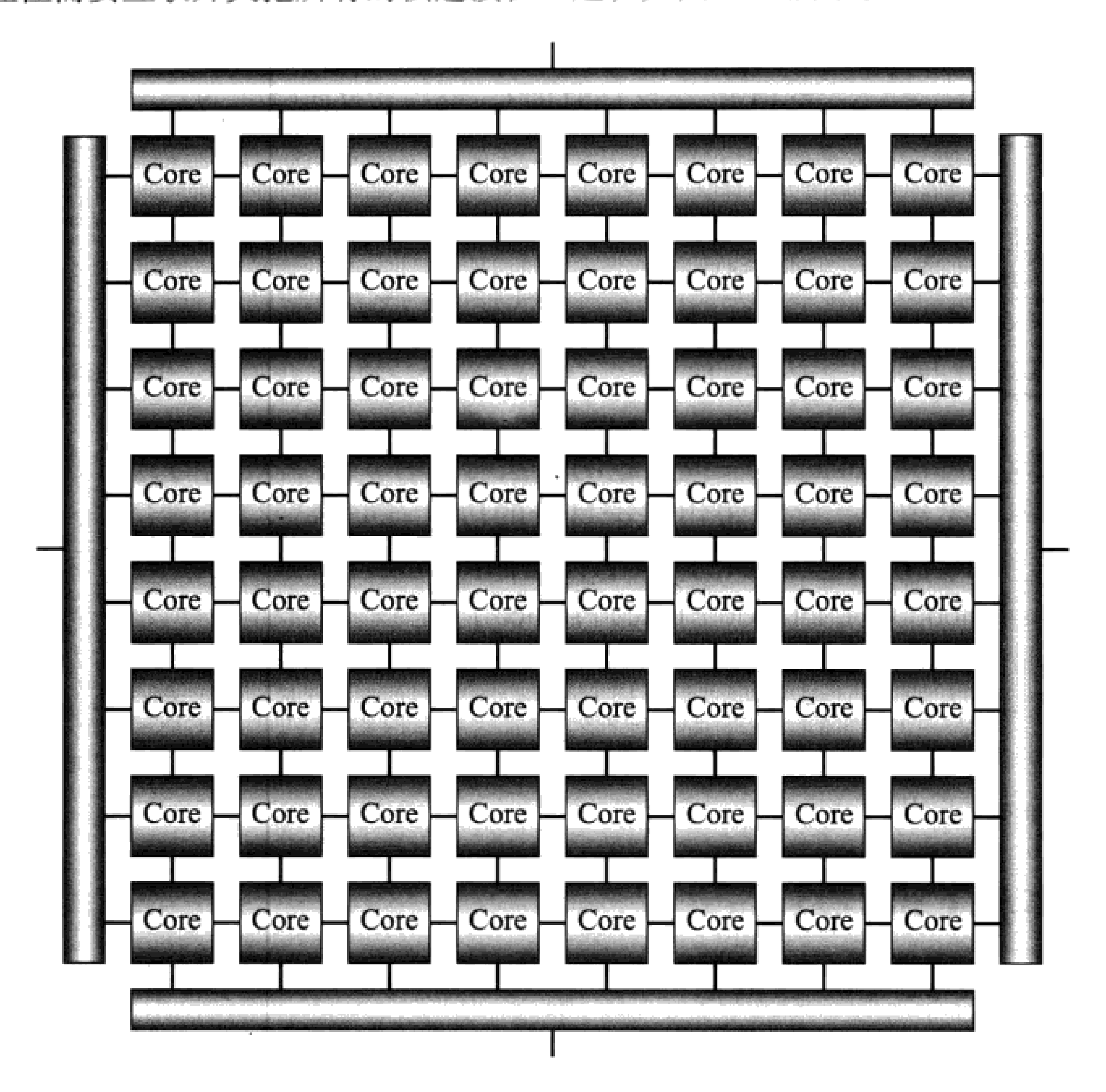


图 14.3 使用互联开关的多核 CPU

图 14.3 中所有的核由 Mesh (一种拓扑结构) 连接在一起，每个核可以有自己的第二级 Cache。当 Cache 不命中时，首先查看其他核的 Cache 中有没有一个备份。如果都没有，才去访问外面的存储器。一般地讲，所有传统意义上的小规模多处理机都可以集成在一个芯片内。Multi-Core 还有另外一个名称：Many-Core。

14.1.2 多核 CPU 的 Cache 一致性问题

多核 CPU 实际上就是一个小规模的多处理机。多处理机的一个显著特点就是共享存储器。而存在于多处理机中的一个著名的问题就是 Cache 一致性 (Cache Coherence) 问题。表 14.1 示出的就是 Cache 一致性问题。假设在 t0 时，存储器 x 单元的内容为 0x55555555。当 Core 1 和 Core 2 读取 x 单元时 (t1 和 t2)，0x55555555 被分别存放在各自的 Cache 中。到这时为止，还没出现任何问题。但在 t3 时，Core 1

往 x 单元写入新的数据 0xaaaaaaaa (假设使用写透策略、也修改了存储器)。此时在 Core 2 Cache 中的 0x55555555 就是已经过时的数据了，即出现了所谓的 Cache 不一致的问题了。

表 14.1　多核 CPU 的 Cache 一致性问题

时间	动 作	Core 1 Cache 的内容	Core 2 Cache 的内容	存储器 x 单元的内容
t0				0x55555555
t1	Core 1 读 x	0x55555555		0x55555555
t2	Core 2 读 x	0x55555555	0x55555555	0x55555555
t3	Core 1 写 x	0xaaaaaaaa	0x55555555	0xaaaaaaaa

多处理机解决 Cache 不一致问题的方法有两种。在大规模的分布式共享存储器的多处理机系统中，往往采用基于目录的协议 (Directory-Based Cache Coherence Protocol)。在小规模的基于总线的共享存储器的多处理机系统中，往往采用总线监视协议 (Bus-Snooping Cache Coherence Protocol)。总线监视协议的内涵正如它的名称所指出的那样，每个 Cache 控制器都要监视总线上的动作。当总线上有写动作发生时，检查自己的 Cache 是否有相同地址的 Cache 块。如果有，则或者把新的数据写入自己相应的 Cache 块 (Update-Based Protocols)，或者干脆把相应的 Cache 块置成无效 (Invalidate-Based Protocols)。

14.2　多核 CPU 设计

14.2.1　多核 CPU 的总体结构

我们的多核 CPU 没有实现 Cache 一致性协议，而且只有一级 Cache。图 14.4 是双核 CPU 的总体结构，共有 4 个 TLB 模块和 4 个 Cache 模块。

图中的 Core 1 模块以及 Core 2 模块与第 13 章的 cpu_cache_tlb 模块相同。关键部分是双核 CPU 与存储器之间的接口。当两个核同时要访问存储器时，必须要有电路解决双核之间的竞争。有了单核，多核 CPU 的设计变简单了。

14.2.2　多核对外部总线的竞争与仲裁

我们令 Core 1 和 Core 2 有平等的机会访问存储器。为此，我们设置了一个一位计数器。当计数器的值为 0 时，Core 1 优先；当计数器的值为 1 时，Core 2 优先。

表 14.2 是产生 select1 信号的真值表。select1 为 1 时选择 Core 1；为 0 时选择 Core 2。表中的 cnt 是一位计数器的输出；mem_access1 和 mem_access2 分别是 Core 1 和 Core 2 的存储器访问请求，为 1 时表示有请求。由此，我们得到 select1 信号的逻辑表达式如下。

```
select1 = ~cnt & mem_access1 | cnt & ~mem_access2;
```

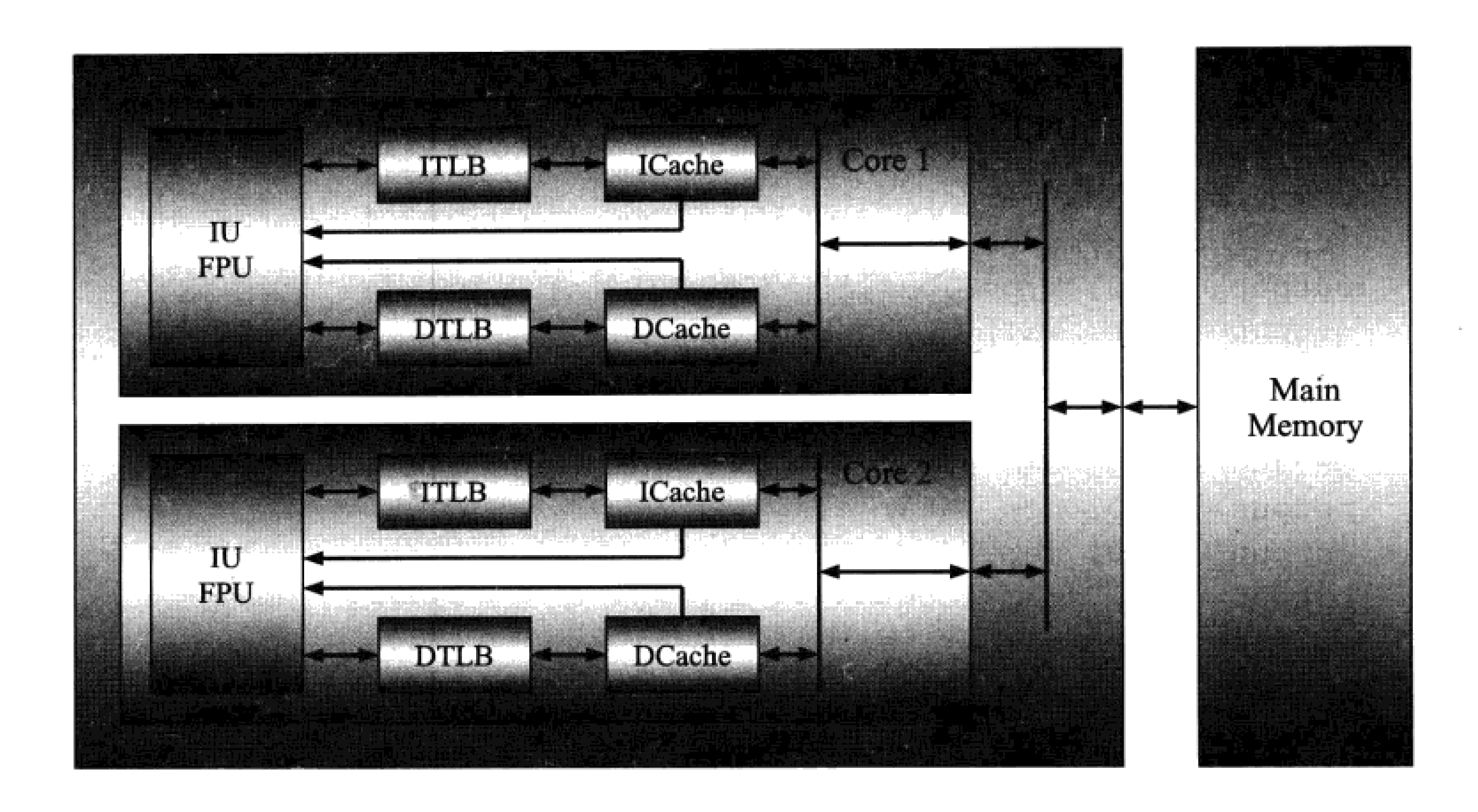


图 14.4 双核 CPU 的总体结构

表 14.2 存储器访问仲裁

cnt	mem_access1	mem_access2	select1
x	0	0	x
0	0	1	0
0	1	0	1
0	1	1	1
1	0	1	0
1	1	0	1
1	1	1	0

有了 select1 信号，我们就可以从两个核的两组存储器访问信号中选择一组 (使用多路器)，送往存储器。同样我们也可以使用 select1 信号把存储器已经准备好的信号送往其中的一个核 (使用分配器)。需要注意的是计数器不能每个时钟周期都计数，而是应该在存储器访问期间停止计数。

有没有办法给 Core 1 更高的优先级来访问存储器呢？有。使用多于一位的计数器，并且按照你自己的喜好随便修改 select1 的真值表。那如果不是双核而是四核或八核怎么办呢？也简单，用两位或三位的选择信号就行。

14.2.3 多核 CPU 的 Verilog HDL 代码

以下是双核 CPU 高层的 Verilog HDL 代码。低层的代码与第 13 章的相同。我们只把 Core 1 的信号放在高层，供测试用。

```
module core2_cache_tlb_memory
   (clock,memclock,resetn,v_pc1,pc1,inst1,ealu1,malu1,walu1,wn1,
    wd1,ww1,stall_lw1,stall_fp1,stall_lwc11,stall_swc11,stall1,
    mem_a,mem_data,mem_st_data,mem_access,mem_write,mem_ready);
    input  clock,memclock,resetn;
    output [31:0] v_pc1,pc1,inst1,ealu1,malu1,walu1;
    output [31:0] wd1;
    output  [4:0] wn1;
    output ww1,stall_lw1,stall_fp1,stall_lwc11,stall_swc11,stall1;
    output [31:0] mem_a;
    output [31:0] mem_data;
    output [31:0] mem_st_data;
    output        mem_access;
    output        mem_write;
    output        mem_ready;
// core1
    wire [31:0] mem_a1;
    wire [31:0] mem_st_data1;
    wire        mem_access1;
    wire        mem_write1;
    cpu_cache_tlb core1
      (clock,memclock,resetn,v_pc1,pc1,inst1,ealu1,malu1,walu1,
       wn1,wd1,ww1,stall_lw1,stall_fp1,stall_lwc11,stall_swc11,
       stall1,mem_a1,mem_data,mem_st_data1,mem_access1,
       mem_write1,mem_ready1);
// core2
    wire [31:0] v_pc2,pc2,inst2,ealu2,malu2,walu2;
    wire [31:0] wd2;
    wire  [4:0] wn2;
    wire ww2,stall_lw2,stall_fp2,stall_lwc12,stall_swc12,stall2;
    wire [31:0] mem_a2;
    wire [31:0] mem_st_data2;
    wire        mem_access2;
    wire        mem_write2;
    cpu_cache_tlb core2
      (clock,memclock,resetn,v_pc2,pc2,inst2,ealu2,malu2,walu2,
       wn2,wd2,ww2,stall_lw2,stall_fp2,stall_lwc12,stall_swc12,
       stall2,mem_a2,mem_data,mem_st_data2,mem_access2,
       mem_write2,mem_ready2);
// mux
    reg cnt;
    always @(negedge resetn or posedge clock) begin
        if (resetn == 0) begin
            cnt <= 0;
        end else if (mem_ready) begin
```

```
            cnt <= ~cnt;
        end
    end
    wire select1 = ~cnt & mem_access1 | cnt & ~mem_access2;
    wire [31:0] mem_a       = select1? mem_a1       : mem_a2;
    wire [31:0] mem_st_data = select1? mem_st_data1 : mem_st_data2;
    wire   mem_access       = select1? mem_access1  : mem_access2;
    wire   mem_write        = select1? mem_write1   : mem_write2;
// demux
    wire   mem_ready1 = mem_ready &  select1;
    wire   mem_ready2 = mem_ready & ~select1;
// main memory
    physical_memory mem (mem_a,mem_data,mem_st_data,mem_access,
                         mem_write,mem_ready,clock,memclock,resetn);
endmodule
```

14.3 多核 CPU 的测试程序及仿真波形

我们使用与第 13 章相同的测试程序，两个核都执行它。仿真波形见图 14.5 ～图 14.14。图中只给出了 Core 1 和与存储器访问有关的信号的波形。请读者参阅第 13 章的测试程序来理解波形的意义。

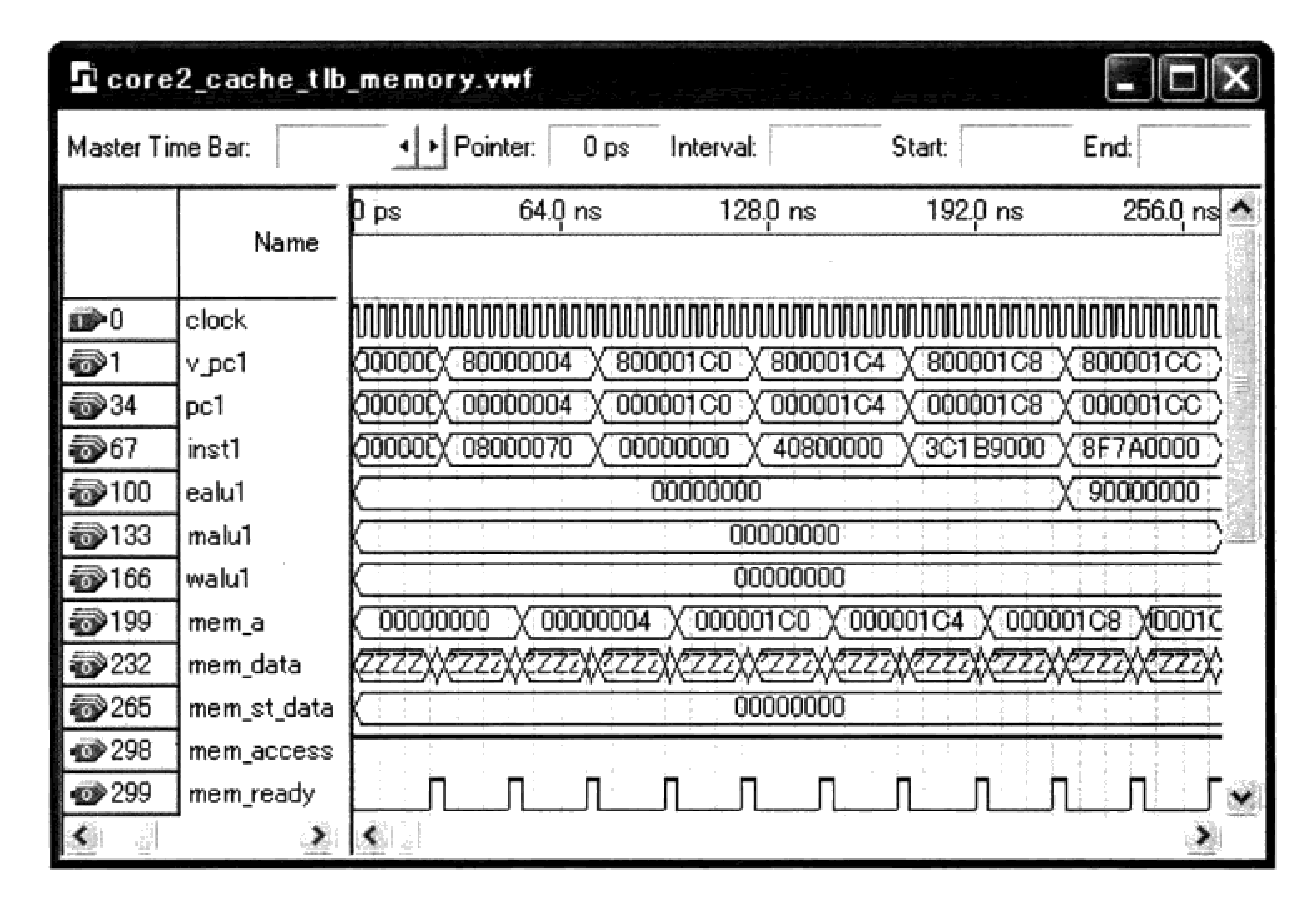


图 14.5 双核 CPU 仿真波形图 (1)

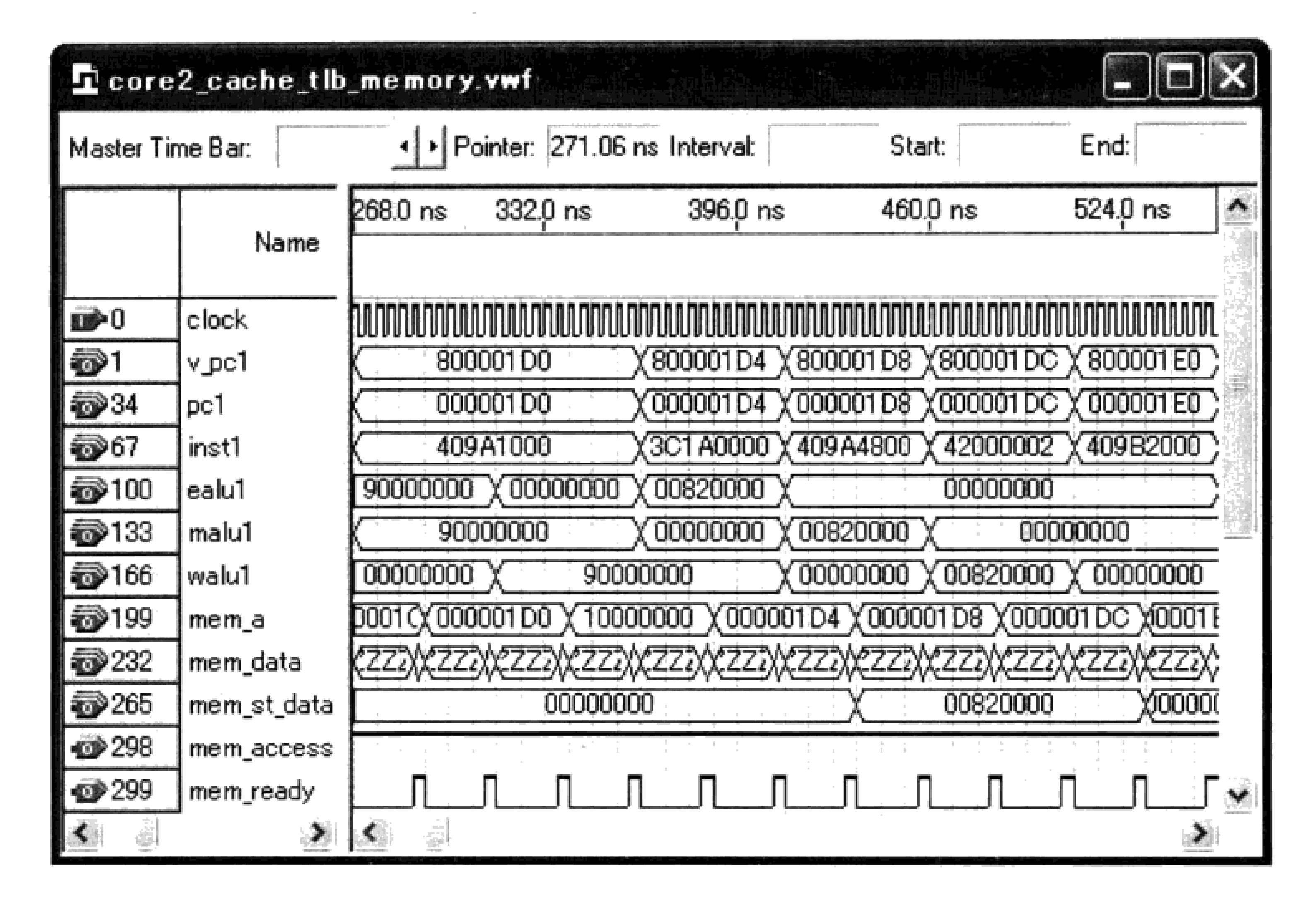


图 14.6 双核 CPU 仿真波形图 (2)

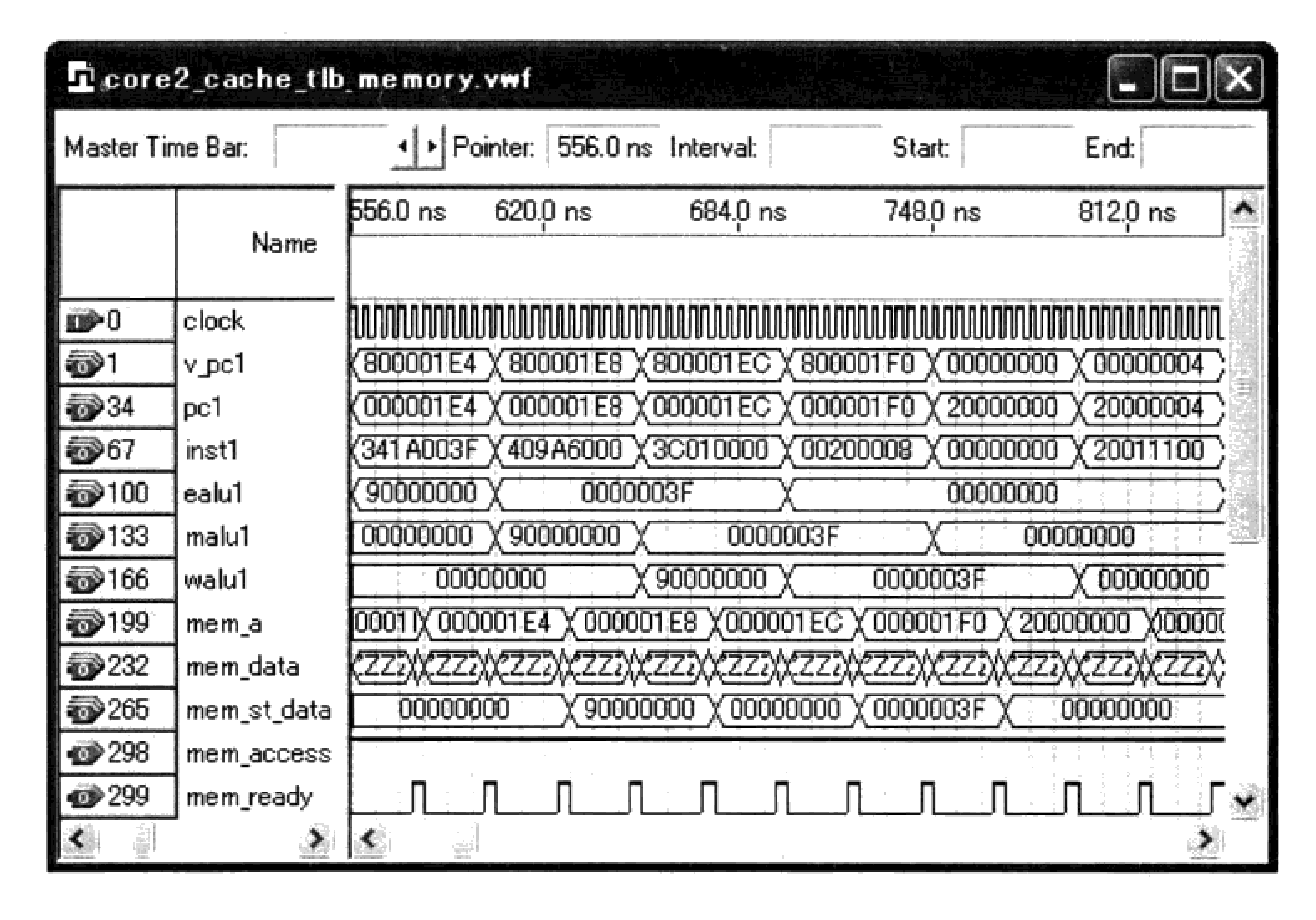


图 14.7 双核 CPU 仿真波形图 (3)

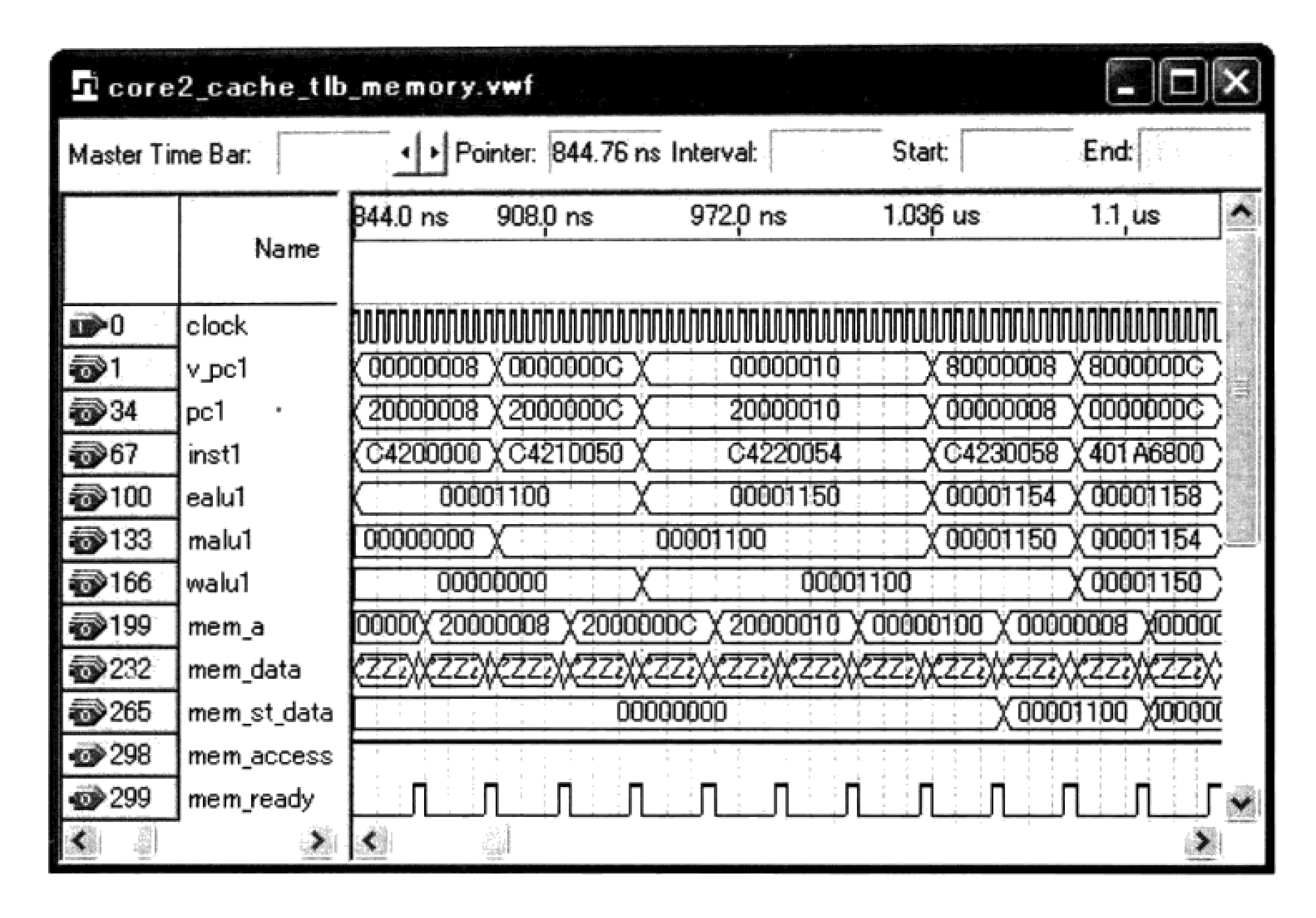


图 14.8 双核 CPU 仿真波形图 (4)

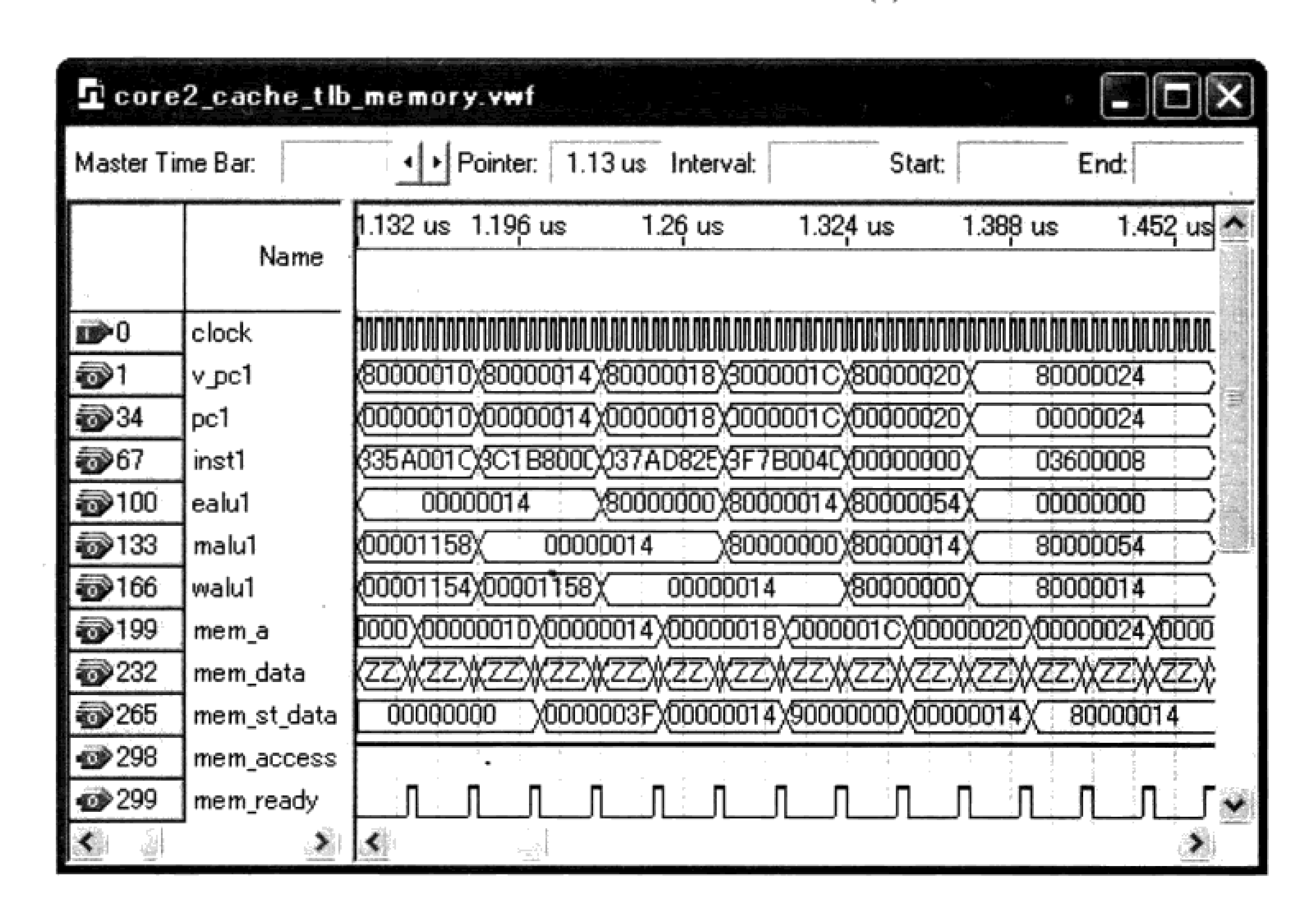


图 14.9 双核 CPU 仿真波形图 (5)

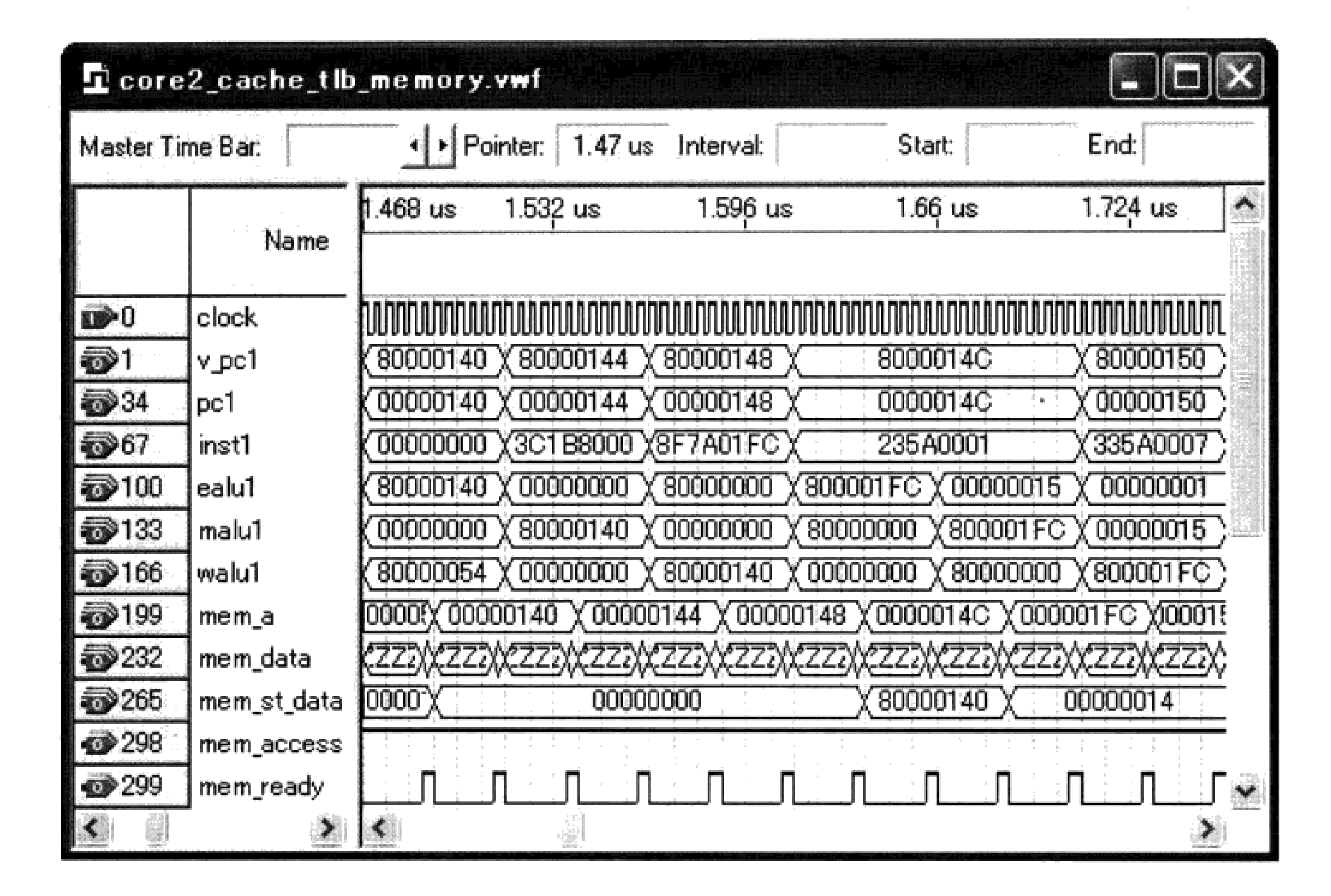


图 14.10 双核 CPU 仿真波形图 (6)

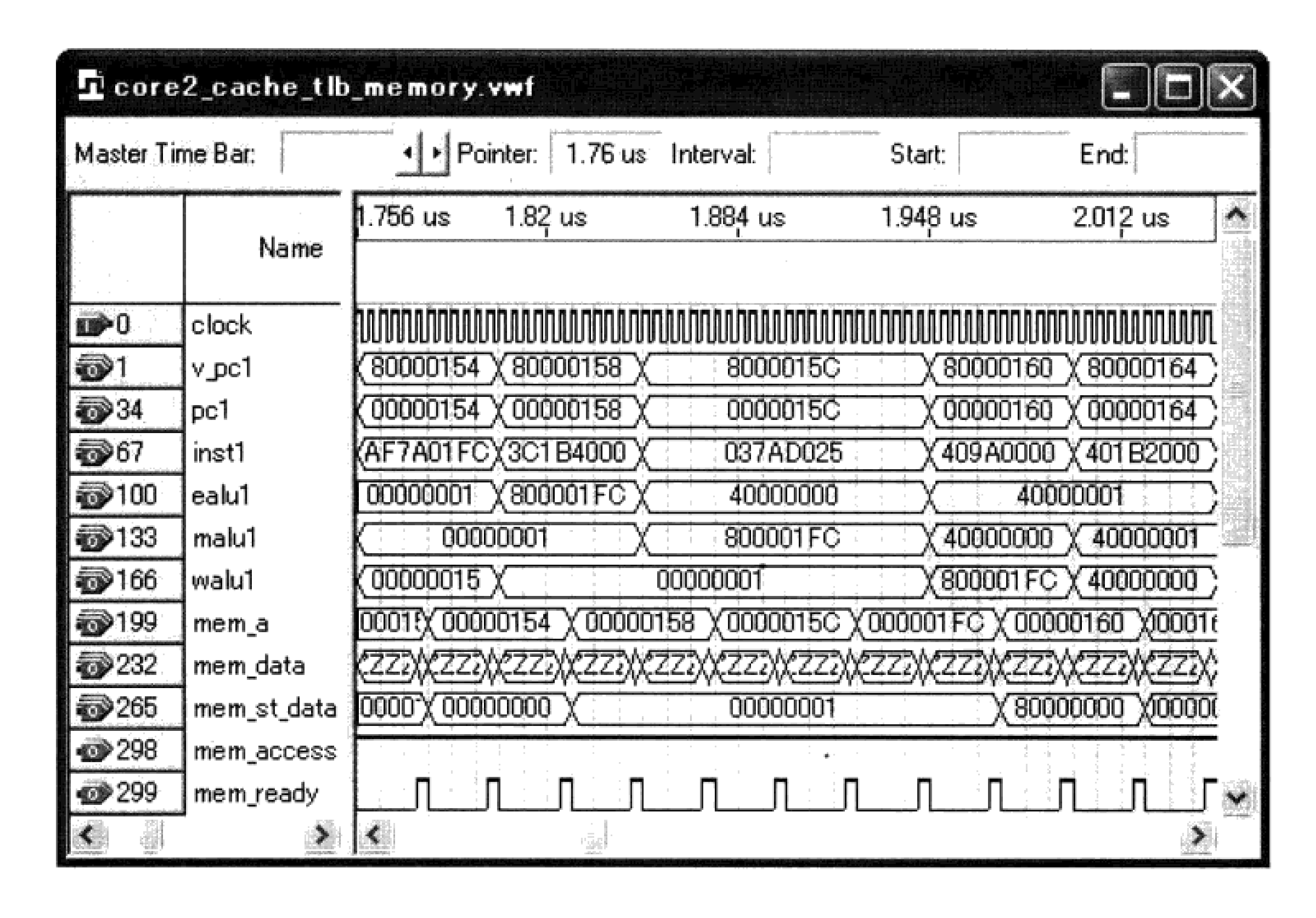


图 14.11 双核 CPU 仿真波形图 (7)

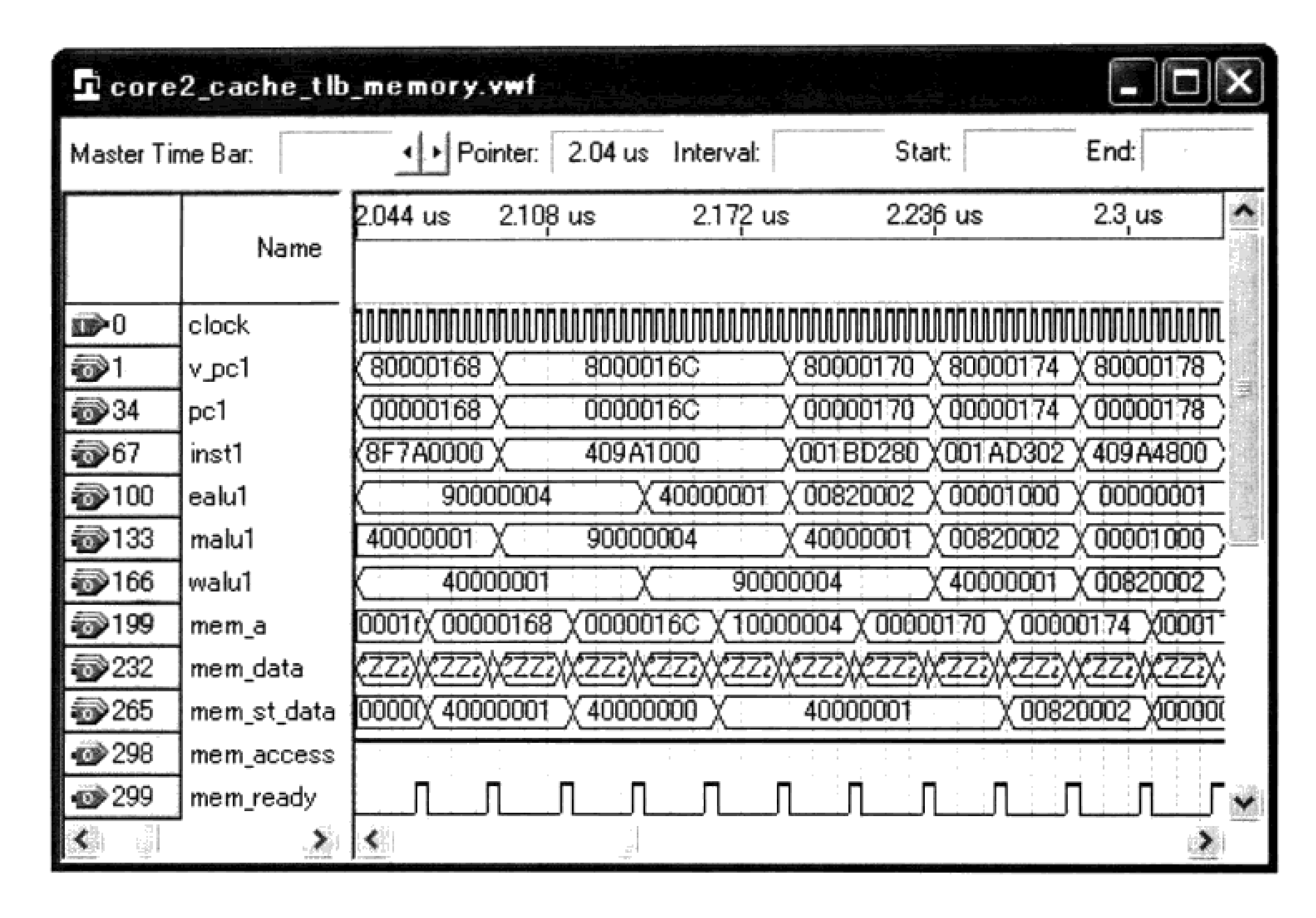


图 14.12　双核 CPU 仿真波形图 (8)

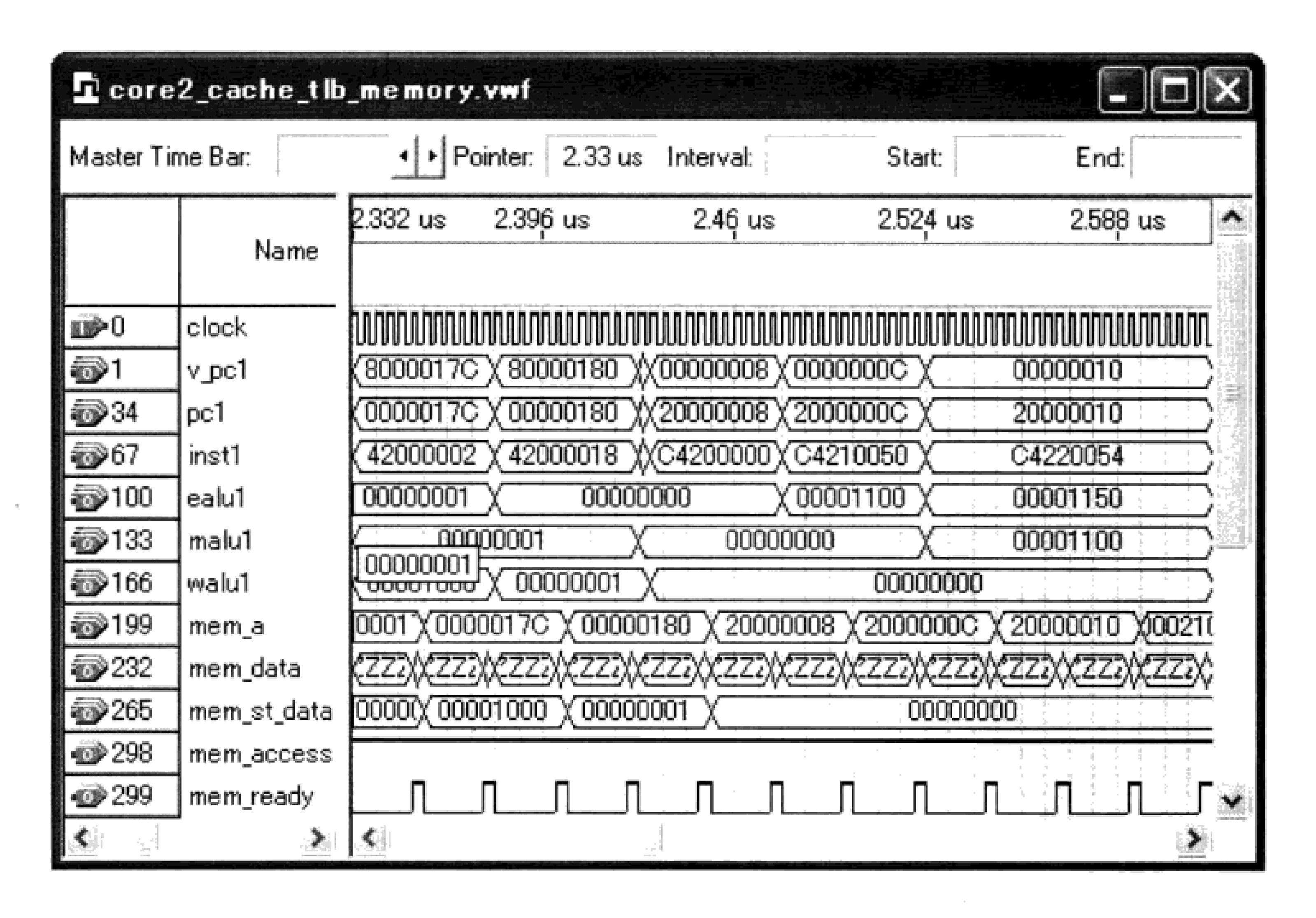


图 14.13　双核 CPU 仿真波形图 (9)

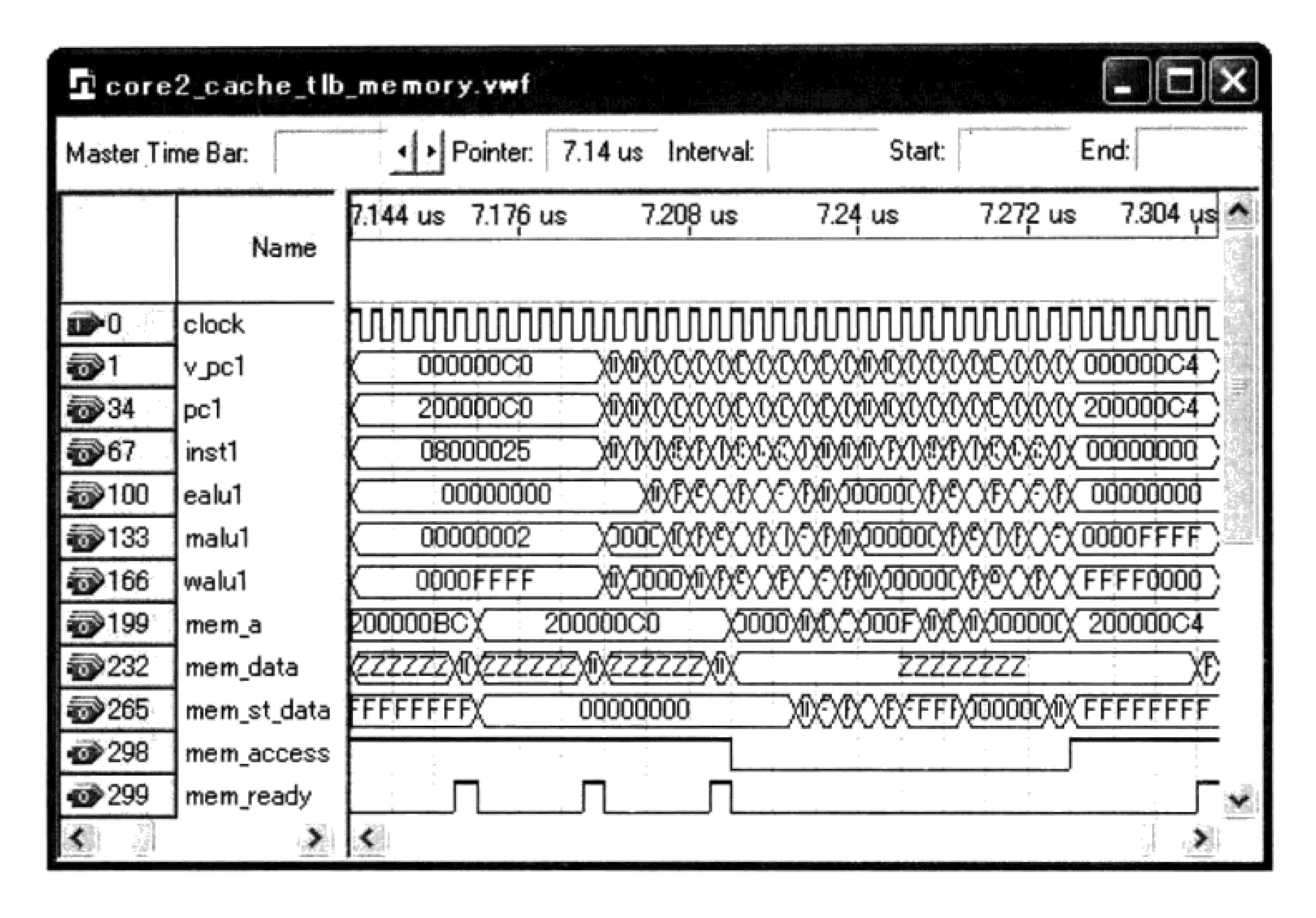


图 14.14　双核 CPU 仿真波形图 (10)

14.4　习题

1. 设计一个双核 CPU 的存储器访问仲裁电路，使 Core 1 有比 Core 2 更高的优先级，比如 5：3。
2. 设计一个只有一级 Cache 的四核 CPU。
3. 设计一个带有第二级 Cache 的双核 CPU (第二级 Cache 共享)。
4. 参考第 11 章，设计一个双核双线程的 CPU，即 CPU 芯片中有两个核，而每个核又是一个双线程的 CPU。

第 15 章　输入/输出接口及设计

本章首先介绍 I/O 接口所涉及的相关技术，然后介绍数据错误的检测及纠正方法，最后介绍几种常用的 I/O 接口和 I/O 总线，包括异步通信接口 UART、PS/2 键盘、PS/2 鼠标、视频图像阵列 VGA、I2C 总线和 PCI 总线。

15.1　I/O 接口概述

I/O 接口连接 CPU 与 I/O 设备，实现二者之间的数据传送。与存储器不同，I/O 设备的速度各种各样。为了能够安全准确地实现 I/O 数据的传送，计算机系统中往往使用专门的总线。另外，为了节省 CPU 时间及加快数据传送速度，CPU 中都有中断及直接存储器访问 (DMA) 机制。

15.1.1　I/O 地址空间和 I/O 指令

依 CPU 不同，I/O 空间有两种不同的实现方法。一种是设置专门的 I/O 空间，与存储器空间并列。访问 I/O 时，使用专门的 I/O 指令，比如 x86 的 in 和 out 指令。另一种是存储器映像的 I/O (Memory Mapped I/O)。这种方式是在虚拟存储器空间中指定一段区域，专门用作 I/O 地址使用。同存储器访问一样，I/O 访问也是使用 Load 和 Store 指令，比如 MIPS。图 15.1 示出了这两种 I/O 空间的结构。

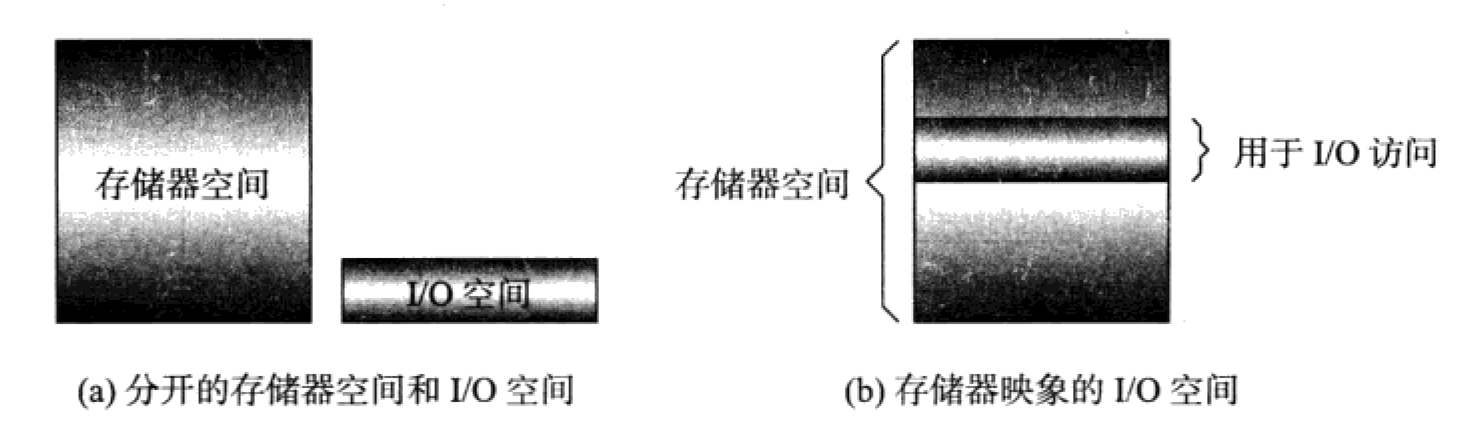


图 15.1　I/O 空间的实现方法

15.1.2　I/O 查询和中断

对 I/O 设备的访问可以通过查询 (Polling) 的方法进行。CPU 从 I/O 状态寄存器读取 I/O 状态，判断 I/O 是否已经准备好。如果还没准备好，继续查询。如果已经准备好，CPU 可以对 I/O 数据进行读写操作，见图 15.2(a)。

查询方式浪费大量的 CPU 时间。为此，在实际的 CPU 设计中，基本上都使用中断 (Interrupt) 方式，见图 15.2(b)。当 I/O 准备好时，由 I/O 接口发出一个中断请求信号给 CPU。CPU 收到这个请求信号后，停止当前程序的运行，转到中断处理程序

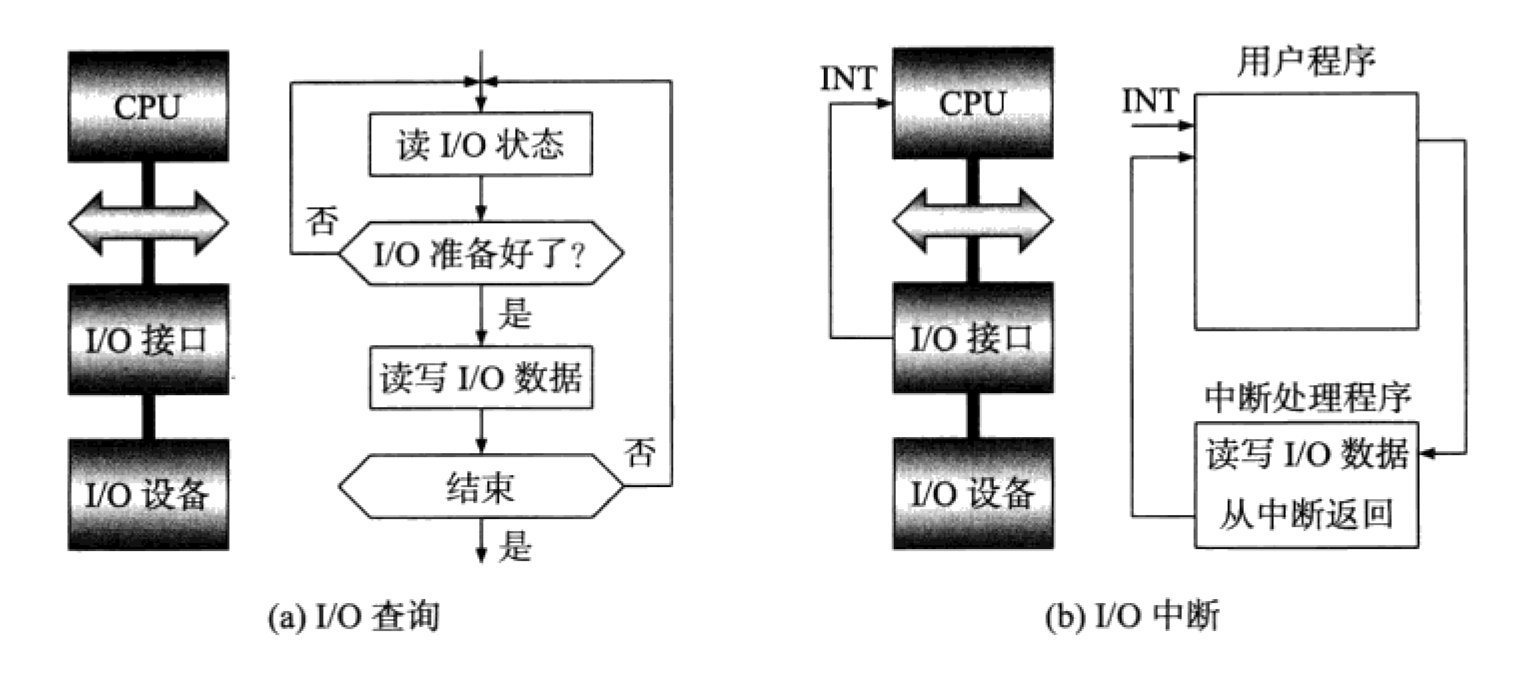


图 15.2 I/O 查询和 I/O 中断

去处理 I/O。处理完毕后，返回到被中断的地方继续执行。我们已经在第 6 章描述了单周期 CPU 处理中断和异常的方法，在第 8 章描述了精确中断的实现方法。

15.1.3 直接存储器访问 DMA

CPU 与 I/O 设备之间的数据传送可以通过执行指令的方法完成。但是，当有大量的数据需要传送时，比如从硬盘把数据读到存储器，这种方法需要花费大量的时间。CPU 的特长是计算，像这种简单的操作可以不用执行指令，完全由硬件负责。这就是所谓的直接存储器访问 (Direct Memory Access)。负责这项工作的硬件称为 DMAC (DMA Controller)，即 DMA 控制器，如图 15.3 所示。

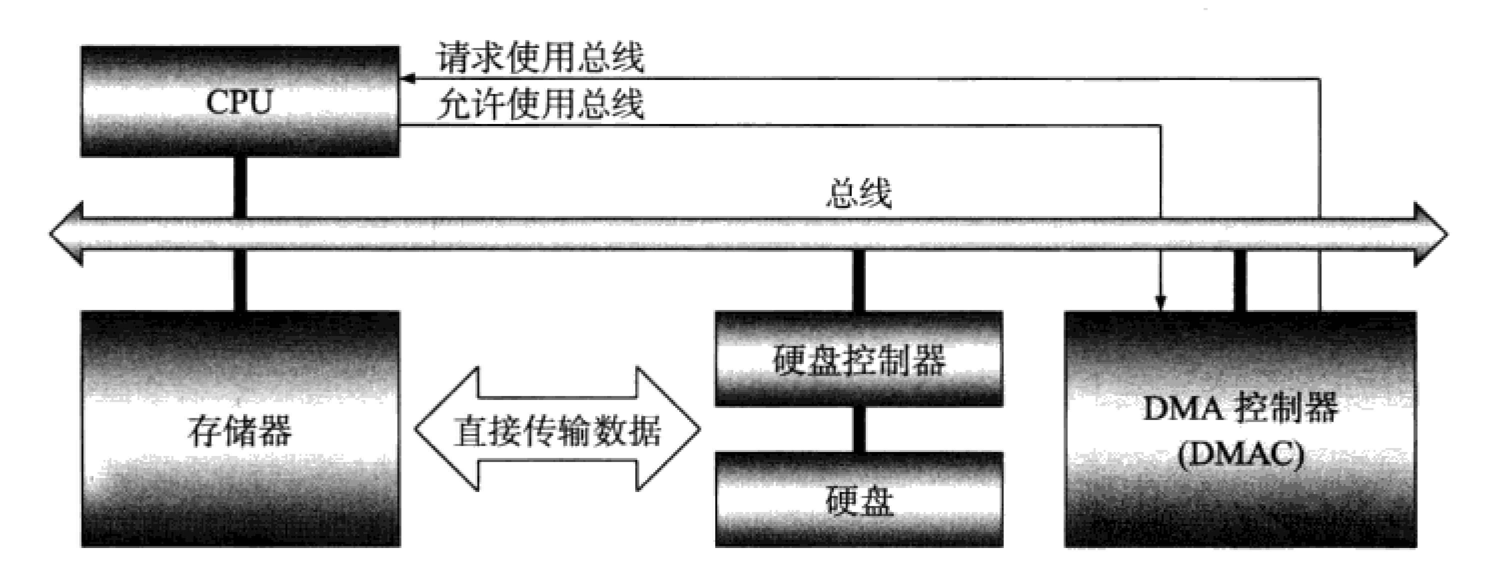


图 15.3 直接存储器访问

为了使 DMAC 能正常工作，CPU 首先要对它初始化，告诉它从哪里传到哪里以及要传多少个字节。然后，DMAC 向 CPU 发出总线请求。CPU 收到总线请求后，释放对总线的控制 (把向总线输出的信号浮空) 并发送响应信号给 DMAC。然后总线由 DMAC 控制，通过硬件时序完成数据的传送。

15.1.4 总线和总线的同步方式

当 I/O 设备的速度比较快且总是能以一定的速度提供或接收数据时，CPU 与 I/O 设备之间的数据传送可以用同步 (Synchronous) 方式进行，见图 15.4(a)。时钟信号 CLK 是数据传送的基准信号。CPU 在时钟上升沿发出读写信号 R/W，写操作时也送出数据。读操作时用时钟上升沿对数据采样。同步方式的优点是速度快且电路简单，缺点是 CPU 与 I/O 设备之间的距离不能太长。

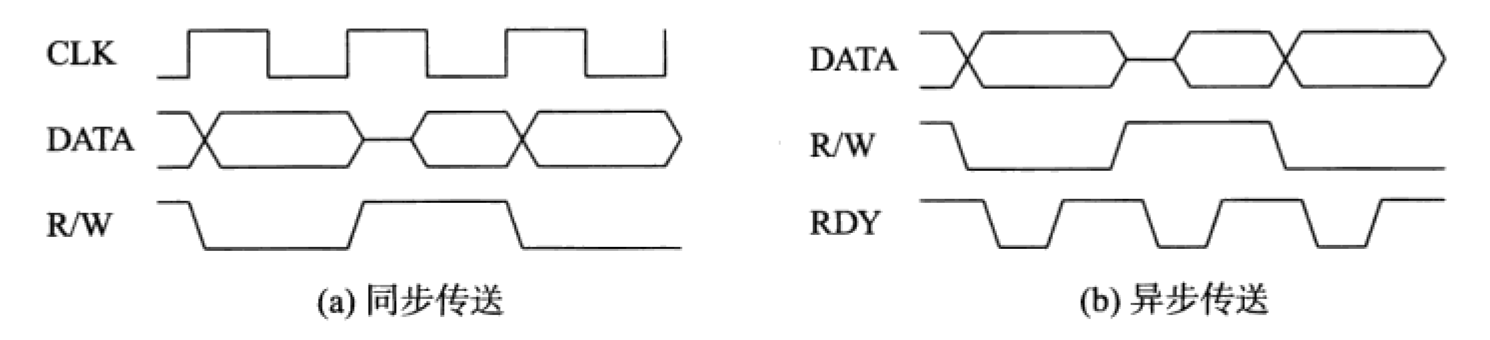


图 15.4 数据传送方式

当 I/O 设备的速度比较慢或距离比较远时，往往采用如图 15.4(b) 所示的异步 (Asynchronous) 方式。在异步方式中没有时钟信号，数据传送的双方用握手信号实现同步，比如图中的准备好信号 RDY。

15.2 数据错误检测及校正

数据在传输过程中或在存储器存放中可能会由于种种原因出现错误。本节介绍两种检测错误的算法 (奇偶校验和循环冗余校验) 和一种纠正一位错误的算法 (扩展的海明码)。

15.2.1 奇偶校验

奇偶校验是为数据加 1 位校验位。数据一般是 8 位，加完校验位后变 9 位。奇偶校验分为奇校验和偶校验。假设我们用 $d_7 \cdots d_0$ 表示 8 位数据，用 p_o 和 p_e 分别表示奇校验位和偶校验位，则有

$$p_o = \overline{d_7 \oplus d_6 \oplus d_5 \oplus d_4 \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0}$$

$$p_e = d_7 \oplus d_6 \oplus d_5 \oplus d_4 \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0$$

即给 8 位任意的数据附加一位 1 或 0，使得奇校验时 9 位数据中为 1 的位数是奇数；使得偶校验时 9 位数据中为 1 的位数是偶数。出错的意思是本来应该为 1 (或 0)，结果却变成了 0 (或 1)。如果数据中出现错误的位数是偶数位的话，不管是奇校验还是偶校验，都查不出来。以两位错为例，因为 $d_i \oplus d_j = \overline{d_i} \oplus \overline{d_j}$，所以查不出来。

换句话说，就是奇偶校验只能检测出有奇数位出错，而不能检测出有偶数位出错。奇偶校验能有效地工作的前提是出现一位错的概率远大于出现两位错的概率。

虽然一位错奇偶校验能检测出来，但不能修复，因为并不知道是哪一位出了错。硬盘阵列 RAID (Redundant Arrays of Inexpensive/Independent Desks) 也用奇偶校验。当有一个硬盘坏了，存于该硬盘的数据能由其他硬盘的数据计算出来。这是因为你已经知道了哪个硬盘坏了，这点和一位数据被搞反了不一样。

15.2.2 错误纠正码 ECC (扩展的海明码)

奇偶校验太简单了。我们希望当有一位错误时，不但能检测出来，而且能够知道是哪一位出的错。这样我们对出错的那一位取反，不就纠正过来了吗？这样的代码称做错误纠正码 ECC (Error Correcting Code)。

为达此目的，我们使用多位校验位，每位校验位只对部分数据位进行校验。我们希望当出现一位错误时 (包括数据位和校验位)，通过计算，能知道出错的位置。海明码 (Hamming Code) 就是干这个用的。以下我们通过一个具体的例子说明海明码是如何工作的。假设有 4 位校验位，11 位数据位。我们把它们排排队：

$$d_{15}\ d_{14}\ d_{13}\ d_{12}\ d_{11}\ d_{10}\ d_9\ p_8\ d_7\ d_6\ d_5\ p_4\ d_3\ p_2\ p_1$$

其中 p_i 是校验位，并且 $i = 2^k$，$0 \leqslant k \leqslant 3$；$d_j$ 是数据位。4 位校验位的产生方法如下：$p_i = \oplus\{d_j\} \mid j \,\&\, i = i$。即

$$\begin{aligned}
p_8 &= d_{15} \oplus d_{14} \oplus d_{13} \oplus d_{12} \oplus d_{11} \oplus d_{10} \oplus d_9 \\
p_4 &= d_{15} \oplus d_{14} \oplus d_{13} \oplus d_{12} \oplus d_7 \oplus d_6 \oplus d_5 \\
p_2 &= d_{15} \oplus d_{14} \oplus d_{11} \oplus d_{10} \oplus d_7 \oplus d_6 \oplus d_3 \\
p_1 &= d_{15} \oplus d_{13} \oplus d_{11} \oplus d_9 \oplus d_7 \oplus d_5 \oplus d_3
\end{aligned}$$

校验时，计算

$$\begin{aligned}
c_8 &= d_{15} \oplus d_{14} \oplus d_{13} \oplus d_{12} \oplus d_{11} \oplus d_{10} \oplus d_9 \oplus p_8 \\
c_4 &= d_{15} \oplus d_{14} \oplus d_{13} \oplus d_{12} \oplus d_7 \oplus d_6 \oplus d_5 \oplus p_4 \\
c_2 &= d_{15} \oplus d_{14} \oplus d_{11} \oplus d_{10} \oplus d_7 \oplus d_6 \oplus d_3 \oplus p_2 \\
c_1 &= d_{15} \oplus d_{13} \oplus d_{11} \oplus d_9 \oplus d_7 \oplus d_5 \oplus d_3 \oplus p_1
\end{aligned}$$

如果没有出错，则 $c = c_8c_4c_2c_1 = 0000$，这是因为 $p \oplus p = 0$。如果有一位出错，则 c 指出出错位的位置。例如 d_{13} 出错，$c = 1101$，这是因为 d_{13} 出现在计算 p_8、p_4 和 p_1 的式子中。我们只要把 d_{13} 取反，错误就被纠正过来了。

以上就是基本的海明码所能完成的任务。如果有两位出错了会出现什么情况呢？我们还是举例来看。假设 d_9 和 d_7 两位错了。由于 d_9 出现在计算 p_8 和 p_1 的式子中，d_7 出现在计算 p_4、p_2 和 p_1 的式子中，会导致校验时计算出的 $c = 1110$，即，$1001 \oplus 0111 = 1110$，指出 d_{14} 出错。如果把 d_{14} 取反，非但没把错误位纠正过来，反而把好端端的一位正确位给糟蹋了。

怎么办？答案是使用扩展的海明码。扩展的海明码 (Extended Hamming Code) 是在海明码的基础上再加一位校验位 p_0，校验时再计算 c_0：

$$p_0 = d_{15} \oplus d_{14} \oplus d_{13} \oplus d_{12} \oplus d_{11} \oplus d_{10} \oplus d_9 \oplus p_8 \oplus d_7 \oplus d_6 \oplus d_5 \oplus p_4 \oplus d_3 \oplus p_2 \oplus p_1$$
$$c_0 = d_{15} \oplus d_{14} \oplus d_{13} \oplus d_{12} \oplus d_{11} \oplus d_{10} \oplus d_9 \oplus p_8 \oplus d_7 \oplus d_6 \oplus d_5 \oplus p_4 \oplus d_3 \oplus p_2 \oplus p_1 \oplus p_0$$

这样，如果还是有两位错，$c \neq 0$ 但 $c_0 = 0$。这时只报告，不纠正。如果只有一位错，$c \neq 0$ 且 $c_0 \neq 0$，纠正。如果有更多位出错会是怎样的情形呢？请读者自己思考。有人把扩展的海明码称做“单纠错双检错”码，从概率的角度考虑大概是对的，但其实不是完全准确的。想想为什么。有了以上公式，写出它们的 Verilog HDL 代码很容易，这里就不再给出了。

15.2.3 循环冗余校验 CRC

循环冗余校验 CRC (Cyclic Redundancy Check) 通常在通信时被用来检查数据是否在传输过程中出错。CRC 码用模 2 的除法产生。设数据有 m 位，它的绝对值为 M。又设有一个 d 位的除数，它的绝对值为 D。发送方把 M 左移 $d-1$ 位再除以 D，得到 $d-1$ 位余数 R。即

$$M \times 2^{d-1} = Q \times D + R$$

其中 Q 为商。注意这里的模 2 除法运算是用异或代替减法。加法也是异或。

发送方把 M 和 R 拼接在一起，即 $N = M \times 2^{d-1} + R$，送给接收方。接收方把 N 除以 D。如果没有出错，余数应为 0。为什么呢？理由如下：

$$M \times 2^{d-1} + R = Q \times D + R + R = Q \times D + 0$$

由于是异或运算，$R + R = 0$。表 15.1 给出一个例子，其中 $D = 1011$，$R = 110$。

表 15.1 CRC 计算举例

发送方			接收方		
	余数高位	R		余数高位	R
M	110001011111	000	M	110001011111	110
⊕	1011		⊕	1011	
=	011101011111	000	=	011101011111	110
⊕	01011		⊕	01011	
=	001011011111	000	=	001011011111	110
⊕	001011		⊕	001011	
=	000000011111	000	=	000000011111	110
⊕	00000001011		⊕	00000001011	
=	000000001001	000	=	000000001001	110
⊕	000000001011		⊕	000000001011	
=	000000000010	000	=	000000000010	110
⊕	000000000010	11	⊕	000000000010	11
=	000000000000	110	=	000000000000	000

异或操作是把余数中对应于除数位是 1 的那些位取反。表中的计算跳过了余数高位的很多 0，直接移到高位是 1 的地方进行异或。由于除数的最高位是 1，异或后相应位的余数就变成了 0。即，余数高位若是 0，跳过 (移位)；若是 1，异或。

我们知道，$x \oplus 0 = x$、$x \oplus 1 = \bar{x}$，相当于每位都做异或操作，只是余数高位是 0 时，异或上全 0；是 1 时，异或上除数。因此我们有图 15.5 所示的电路。图中有三个 D 触发器保存余数。本例中除数 (1011) 有 3 位 1，但高位异或要么是 $1 \oplus 1 = 0$，要么就是 $0 \oplus 0 = 0$，总之结果是 0，没必要运算，因此只需要两个异或门。该电路与用加法器和异或门做减法的电路有异曲同工之处，读者不妨比较一番。

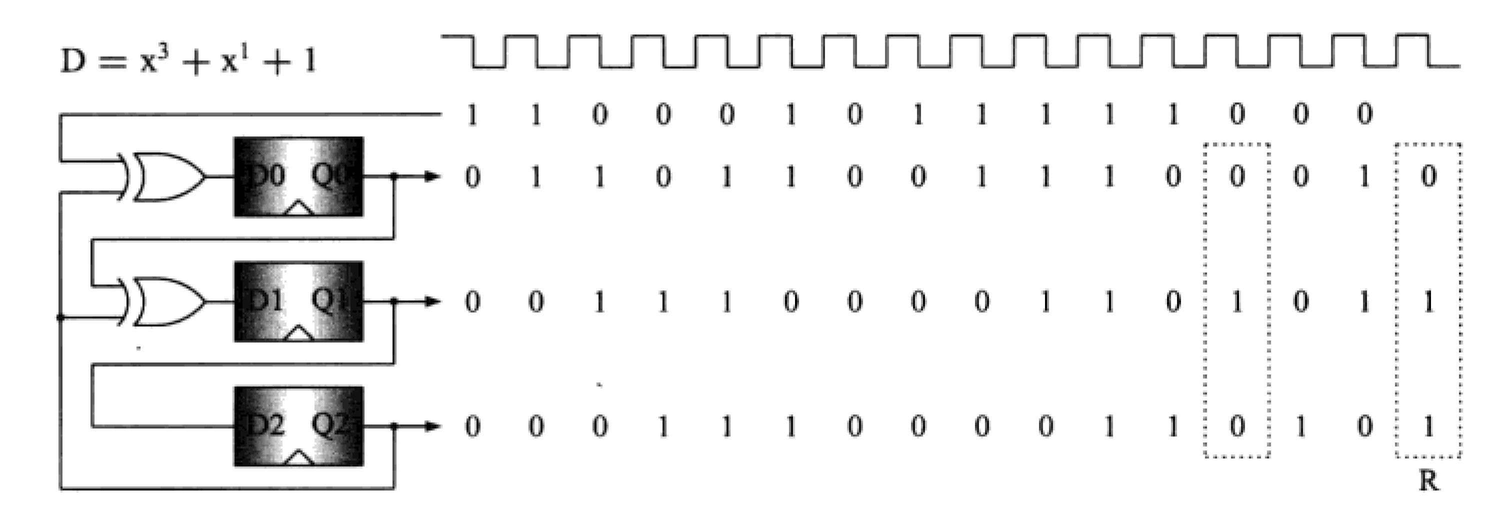


图 15.5 由 $x^3 + x^1 + 1$ 产生 R 的电路

实际上最后的 $d - 1$ 次异或运算不需要，直接把 $R = 010$ 送出。接收方做同样的运算，比较计算出的余数是否与接收到的余数相等，以判断数据在传输过程中是否出错。常用的 CRC 生成多项式 (相应的系数构成除数) 列于表 15.2 中。

表 15.2 常用的 CRC 生成多项式

CRC-12	$x^{12}+x^{11}+x^3+x^2+x+1$	CRC-16-CCITT	$x^{16}+x^{12}+x^5+1$
CRC-16-IBM	$x^{16}+x^{15}+x^2+1$	CRC-64-ISO	$x^{64}+x^4+x^3+x+1$
CRC-32-IEEE	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$		

15.3 异步通信接口 UART

异步通信接口 UART (Universal Asynchronous Receiver Transmitter) 能够发送和接收串行数据。异步的意思是通信的双方之间没有用于同步的时钟信号线。假设以字符为单位发送和接收串行数据，字符与字符之间要有明显的标志把两个字符区分开来。异步通信接口的数据帧 (Data Frame) 格式如图 15.6 所示。

一个数据帧包含有一位起始 (Start) 位，5 至 8 位数据位，1 位奇偶校验 (Parity) 位 (可有可无) 和 1 至 2 位停止 (Stop) 位。起始位为低电平，停止位为高电平，数据按最低位 LSB (Least Significant Bit) 到最高位 MSB (Most Significant Bit) 的次序送出

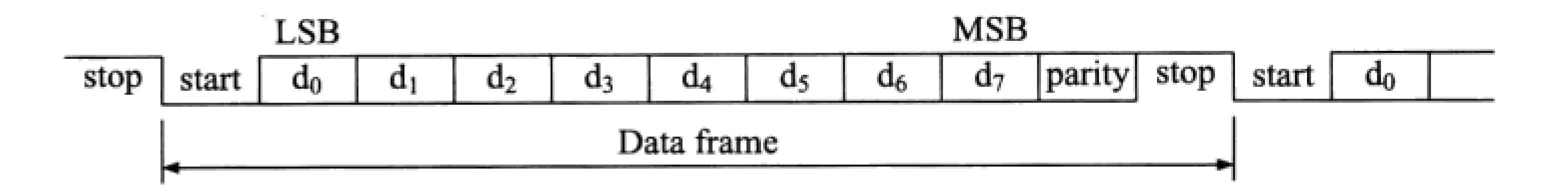


图 15.6 异步通信接口的数据帧格式

或到达。图中及以下的 Verilog HDL 代码假定数据有 8 位、偶校验。图 15.7 示出的是 UART 的信号及连接方式，将在后续的描述中给出各信号的意义。

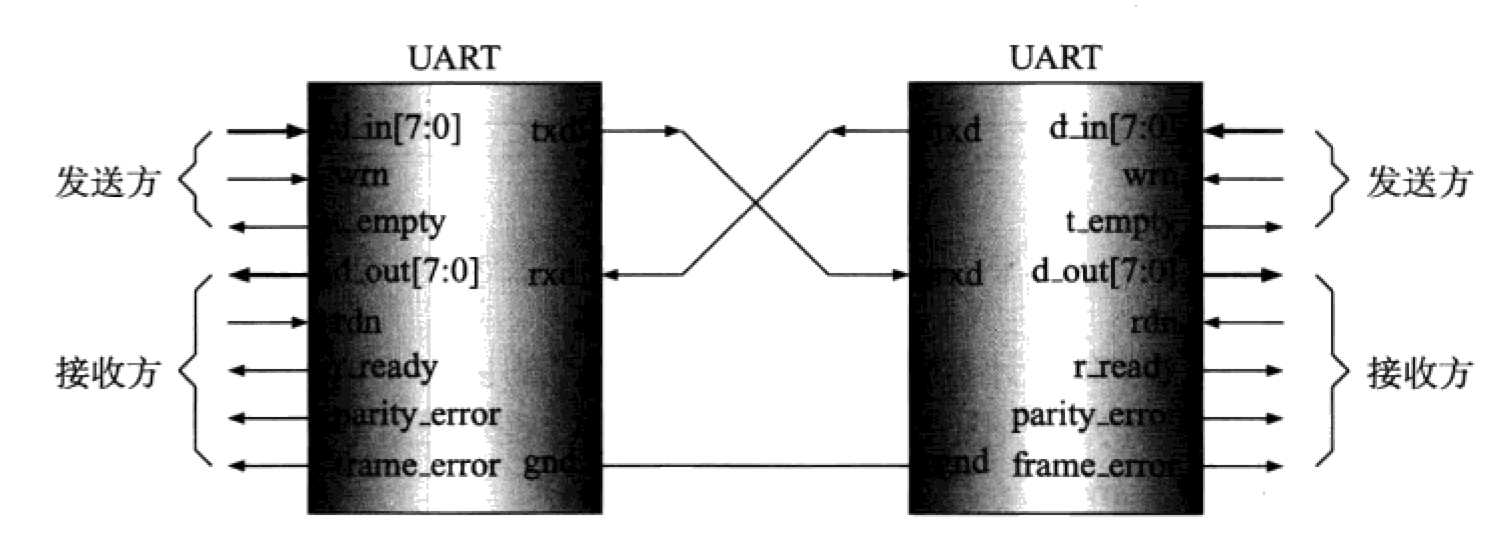


图 15.7 异步通信接口的信号及连接方式

实现异步通信的双方必须事先约定好所谓的波特率 (Baud Rate)，即每秒传送的二进制位数，包含起始位、校验位和停止位。常用的波特率有 4800、9600 和 19200。由于没有时钟信号，接收方的电路必须要检测起始位，然后按波特率来采样剩余的数据。

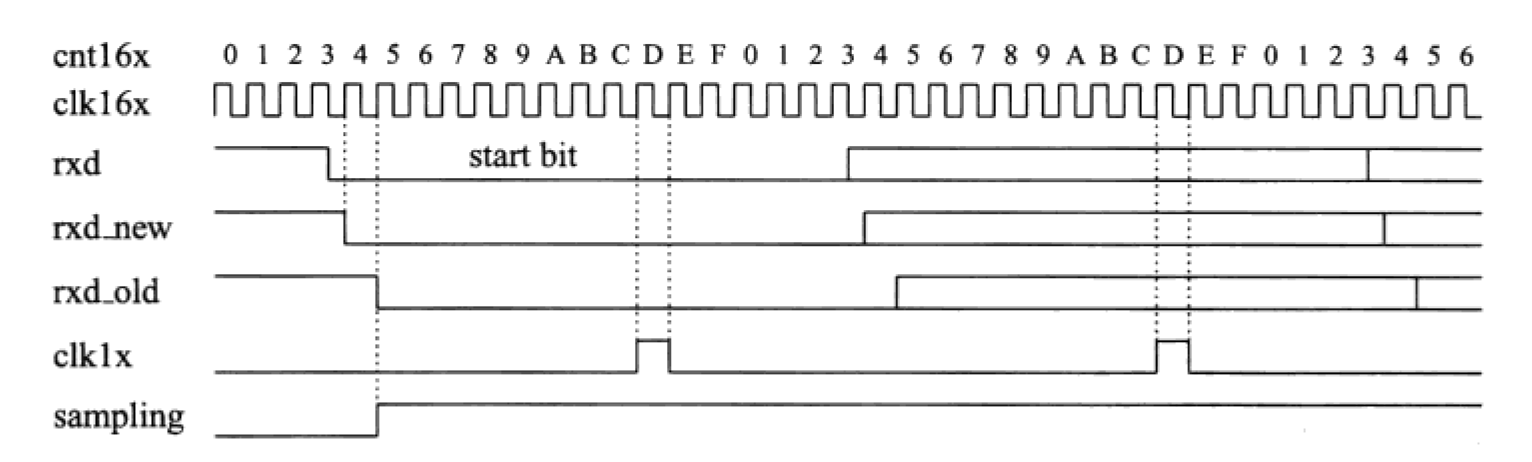


图 15.8 异步通信接口的接收时序

图 15.8 示出的是检测起始位时的波形。我们使用一个内部时钟信号 clk16x，它的频率被设定为波特率的 16 倍。一个 4 位的计数器用来对 clk16x 分频，产生采样脉冲。接收数据线 rxd 上的起始位可以出现在任何时刻。我们用 clk16x 来检测它的下降沿。在 clk16x 的上升沿处，两个信号 rxd_old 和 rxd_new 分别用来保存 rxd_new 和 rxd (需要两个 DFF)。因此，当 rxd_old 为高电平且 rxd_new 为低电平时，我们知道起

始位已经到来。然后令采样的使能信号 sampling 为 1，并在起始位的中间的位置产生采样脉冲 clk1x。采样脉冲高电平的时间长度等于 clk16x 时钟的一个周期。

为了能够知道在何时已经接收到一个完整的数据帧，我们必须要有一个计数器来记录已经接收了多少位，如图 15.9 所示。计数器命名为 no_bits_rcvd。

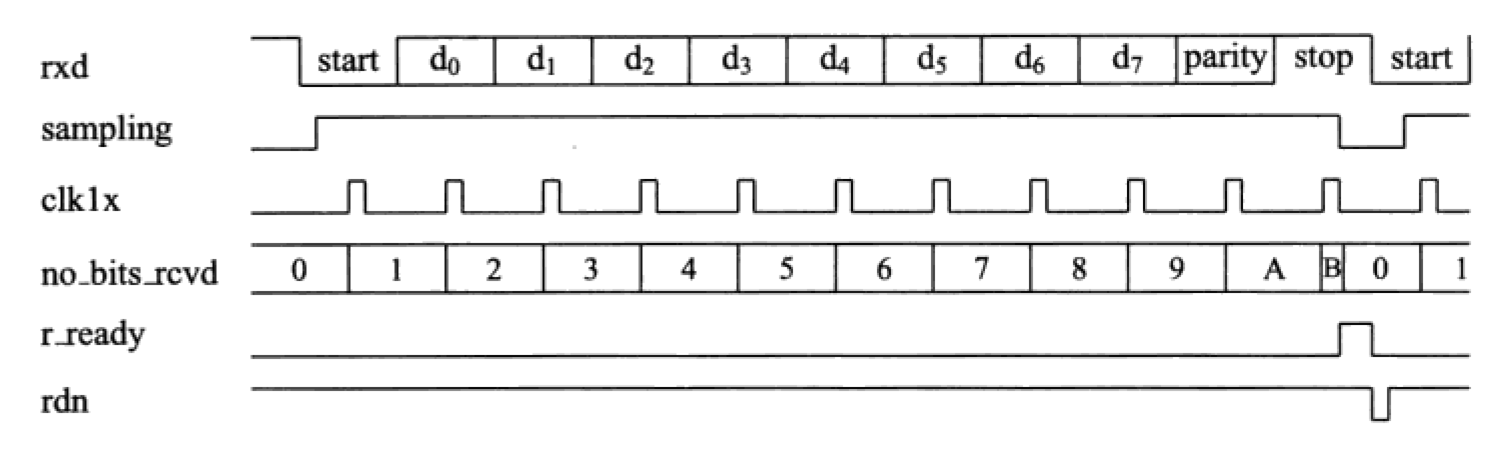


图 15.9 异步通信接口接收一个数据帧

当接收器开始采样时，计数器开始计数。计数器的值为 1 时，表示已经接收到了起始位；计数器的值为 11 (图中十六进制的 B) 时，表示已经接收到了停止位。这时要把采样使能信号 sampling 置为 0。

我们的接收器允许接收连续的数据帧。如果接收到的一帧数据没被及时读走，新来的数据有可能冲掉原来的数据。为了不使已经接收到的数据丢失，我们使用了双缓冲器 r_buffer 和 frame。r_buffer 用于接收 rxd，frame 用于保存已经接收到的 8 位数据。当 r_ready 信号为 1 时，接收方可以使用 rdn (低电平有效的读信号) 从 frame 中读取 8 位数据。当然，如果一直不读走，再多的缓冲器也不够用。接收器也检查校验位，如果出错，把 parity_error 置 1；如果没有接收到停止位，把 frame_error 置 1。

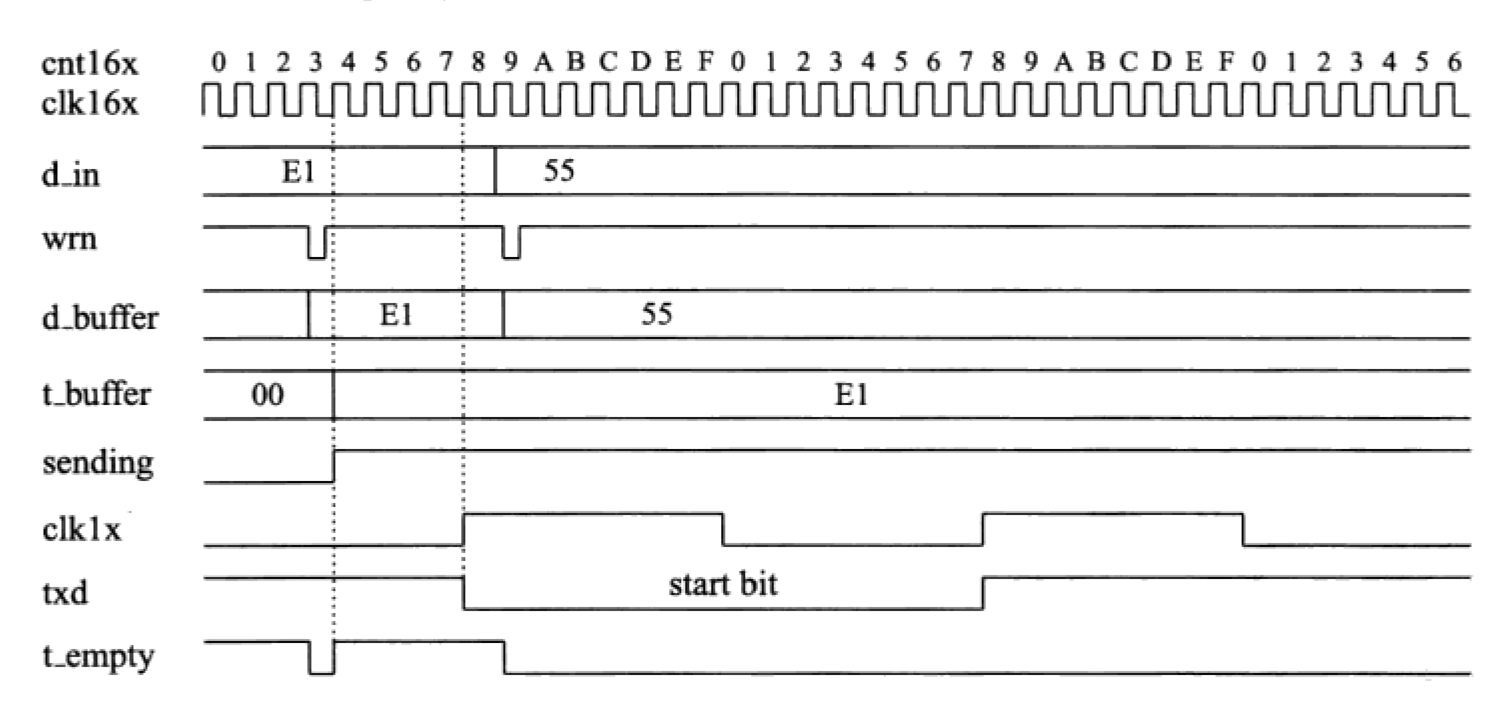


图 15.10 异步通信接口的发送时序

UART 是 “全双工” (Full-Duplex) 的，即接收和发送可以同时进行。发送时序如图 15.10 所示。为了能够实现数据帧的连续发送，我们在发送器中也设置了双缓冲

器：发送方使用 wrn (低电平有效的写信号) 把 d_in 写入 d_buffer，然后把 d_buffer 写入 t_buffer (供发送用)。t_empty 信号为 1 时，表示发送方可以向 d_buffer 写入新的数据，尽管前一个数据的发送还没有结束。与接收器不同，发送器的 clk1x 时钟信号只是对 clk16x 的简单的分频。sending 是发送使能信号。

同样，我们也为发送方设置一个计数器 no_bits_sent，用来控制何时送出校验位和停止位，如图 15.11 所示。由于有了双缓冲器，发送器可以不间断地发送数据帧。

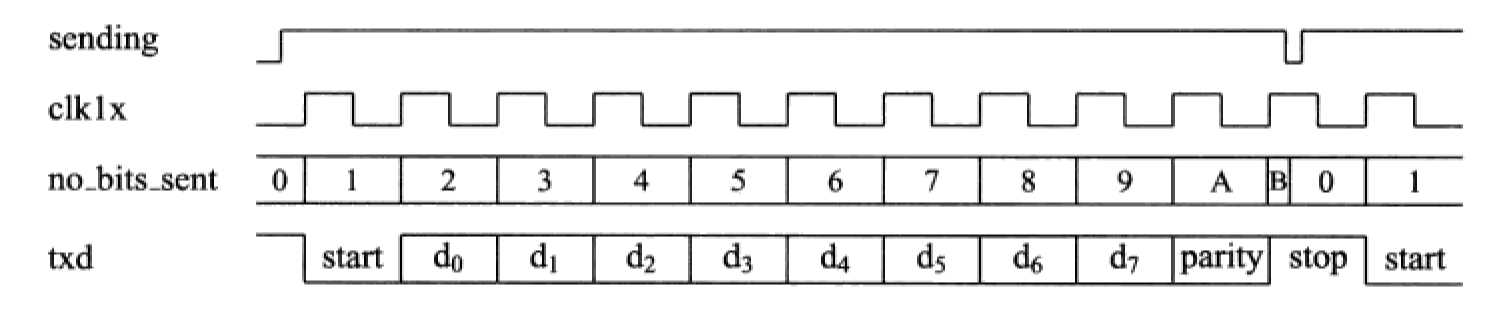


图 15.11　异步通信接口发送一个数据帧

以下是 UART 的 Verilog HDL 代码。注意，为了测试 UART 的工作状态，我们把一些重要的内部信号也拉到了外面。实际的输入输出信号见图 15.7。

```
module uart (clk16x, clrn,
             rdn, d_out, r_ready, rxd, parity_error, frame_error,
             wrn, d_in,  t_empty, txd, cnt16x,
             no_bits_rcvd, r_buffer, r_clk1x, sampling, frame,
             no_bits_sent, t_buffer, t_clk1x, sending, d_buffer);
    input  clk16x, clrn;
    // for receiver
    input  rdn;
    input  rxd;
    output r_ready;
    output [7:0] d_out;
    output parity_error;
    output frame_error;
    // for transmitter
    input  wrn;
    output txd;
    output t_empty;
    input  [7:0] d_in;
    // for test (internal signals)
    output [3:0] cnt16x;
    output sampling;
    output r_clk1x;
    output [3:0] no_bits_rcvd;
    output [10:0] r_buffer;
    output [7:0] frame;
    output sending;
    output t_clk1x;
```

```
    output [3:0] no_bits_sent;
    output [7:0] t_buffer;
    output [7:0] d_buffer;
    // clk16x counter
    reg [3:0] cnt16x;
    always @ (posedge clk16x or negedge clrn) begin
        if (clrn == 0) begin
            cnt16x <= 4'b0000;
        end else begin
            cnt16x <= cnt16x + 4'b0001;
        end
    end
    uart_rx r (clk16x,clrn,rdn,d_out,r_ready,rxd,parity_error,
               frame_error,cnt16x,frame,no_bits_rcvd,
               r_buffer,r_clk1x,sampling);
    uart_tx t (clk16x,clrn,wrn,d_in, t_empty,txd,cnt16x,
               no_bits_sent,t_buffer,t_clk1x,sending,d_buffer);
endmodule

// receiver
module uart_rx (clk16x,clrn,rdn,d_out,r_ready,rxd,parity_error,
                frame_error,cnt16x,frame,no_bits_rcvd,
                r_buffer,clk1x,sampling);
    input  clk16x,clrn;
    input  rdn;
    input  rxd;
    output reg r_ready;
    output [7:0] d_out;
    output reg parity_error;
    output reg frame_error;
    input  [3:0] cnt16x;
    // for test
    output [3:0] no_bits_rcvd;
    output [10:0] r_buffer;
    output [7:0] frame;
    output clk1x;
    output sampling;
    // internal signals
    reg [3:0] sampling_place;
    reg [3:0] no_bits_rcvd;
    reg [10:0] r_buffer; // stop,parity,data[7:0],start
    reg clk1x;
    reg rxd_old,rxd_new;
    reg sampling;
    reg [7:0] frame;
```

```
// latch 2 sampling bits
always @ (posedge clk16x or negedge clrn) begin
    if (clrn == 0) begin
        rxd_old <= 1'b1;
        rxd_new <= 1'b1;
    end else begin
        rxd_old <= rxd_new;
        rxd_new <= rxd;
    end
end
// detect start bit
always @ (posedge clk16x or negedge clrn) begin
    if (clrn == 0) begin
        sampling <= 1'b0;
    end else begin
        if (rxd_old && !rxd_new) begin
            if (!sampling)
                sampling_place <= cnt16x + 4'b1000;
            sampling <= 1'b1;
        end else begin
            if (no_bits_rcvd == 4'b1011)
                sampling <= 1'b0;
        end
    end
end
// sampling clock: clk1x
always @ (posedge clk16x or negedge clrn) begin
    if (clrn == 0) begin
        clk1x <= 1'b0;
    end else begin
        if (sampling) begin
            if (cnt16x == sampling_place)
                clk1x <= 1'b1;
            if (cnt16x == sampling_place + 4'b0001)
                clk1x <= 1'b0;
        end else clk1x <= 1'b0;
    end
end
// number of bits received
always @ (posedge clk1x or negedge sampling) begin
    if (!sampling) begin
        no_bits_rcvd <= 4'b0000;
    end else begin
        no_bits_rcvd <= no_bits_rcvd + 4'b0001;
        r_buffer[no_bits_rcvd] <= rxd;
```

```
        end
    end
    // one frame, rdn clears r_ready
    always @ (posedge clk16x or negedge clrn or negedge rdn) begin
        if (clrn == 0) begin
            r_ready <= 1'b0;
            parity_error <= 1'b0;
            frame_error <= 1'b0;
        end else begin
            if (!rdn) begin
                r_ready <= 1'b0;
                parity_error <= 1'b0;
                frame_error <= 1'b0;
            end else begin
                if (no_bits_rcvd == 4'b1011) begin
                    frame <= r_buffer[8:1];
                    r_ready <= 1'b1;
                    if (^r_buffer[9:1]) begin
                        parity_error <= 1'b1;
                    end
                    if (!r_buffer[10]) begin
                        frame_error <= 1'b1;
                    end
                end
            end
        end
    end
    assign d_out = !rdn ? frame : 8'bz ;
endmodule

// transmitter
module uart_tx (clk16x,clrn,wrn,d_in, t_empty,txd,cnt16x,
                no_bits_sent,t_buffer,clk1x,sending,d_buffer);
    input  clk16x,clrn;
    input  wrn;
    output reg txd;
    output reg t_empty;
    input  [7:0] d_in;
    input  [3:0] cnt16x;
    // for test
    output [3:0] no_bits_sent;
    output [7:0] t_buffer;
    output clk1x;
    output sending;
    output [7:0] d_buffer;
```

```
// internal signals
reg [3:0] no_bits_sent;
reg [7:0] t_buffer;
reg sending;
reg [7:0] d_buffer;
reg load_t_buffer;
// load d_in, sending enable, t_empty, sending_place
always @ (posedge clk16x or negedge clrn or negedge wrn) begin
    if (clrn == 0) begin
        sending <= 1'b0;
        t_empty <= 1'b1;
        load_t_buffer <= 1'b0;
    end else begin
        if (!wrn) begin // only happend in t_empty == 1'b1;
            d_buffer <= d_in;
            t_empty <= 1'b0;
            load_t_buffer <= 1'b1;
        end else begin
            if (!sending) begin
                if (load_t_buffer) begin
                    sending <= 1'b1;
                    t_buffer <= d_buffer;
                    t_empty <= 1'b1;
                    load_t_buffer <= 1'b0;
                end
            end else begin
                if (no_bits_sent == 4'b1011)
                    sending <= 1'b0;
            end
        end
    end
end
assign clk1x = cnt16x[3];
// number of bits sent
always @ (posedge clk1x or negedge sending) begin
    if (!sending) begin
        no_bits_sent <= 4'b0000;
        txd <= 1'b1;
    end else begin
        case (no_bits_sent)
            0: txd <= 1'b0; // start bit
            1: txd <= t_buffer[0];
            2: txd <= t_buffer[1];
            3: txd <= t_buffer[2];
            4: txd <= t_buffer[3];
```

```
                5: txd <= t_buffer[4];
                6: txd <= t_buffer[5];
                7: txd <= t_buffer[6];
                8: txd <= t_buffer[7];
                9: txd <= ^t_buffer; // parity bit
                default: txd <= 1'b1; // stop bit(s)
            endcase
            no_bits_sent <= no_bits_sent + 4'b0001;
        end
    end
endmodule
```

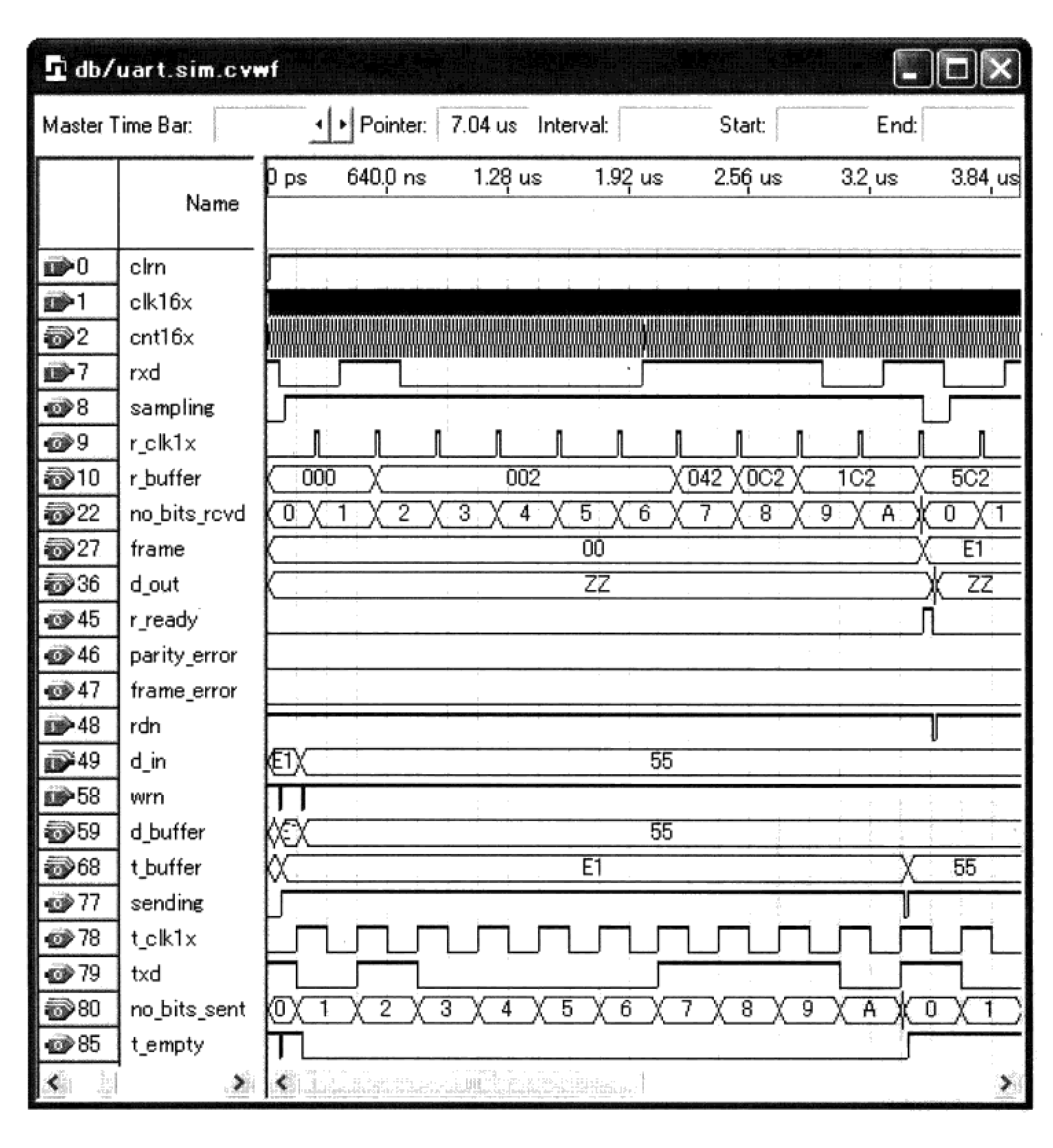


图 15.12 UART 仿真波形 (1)

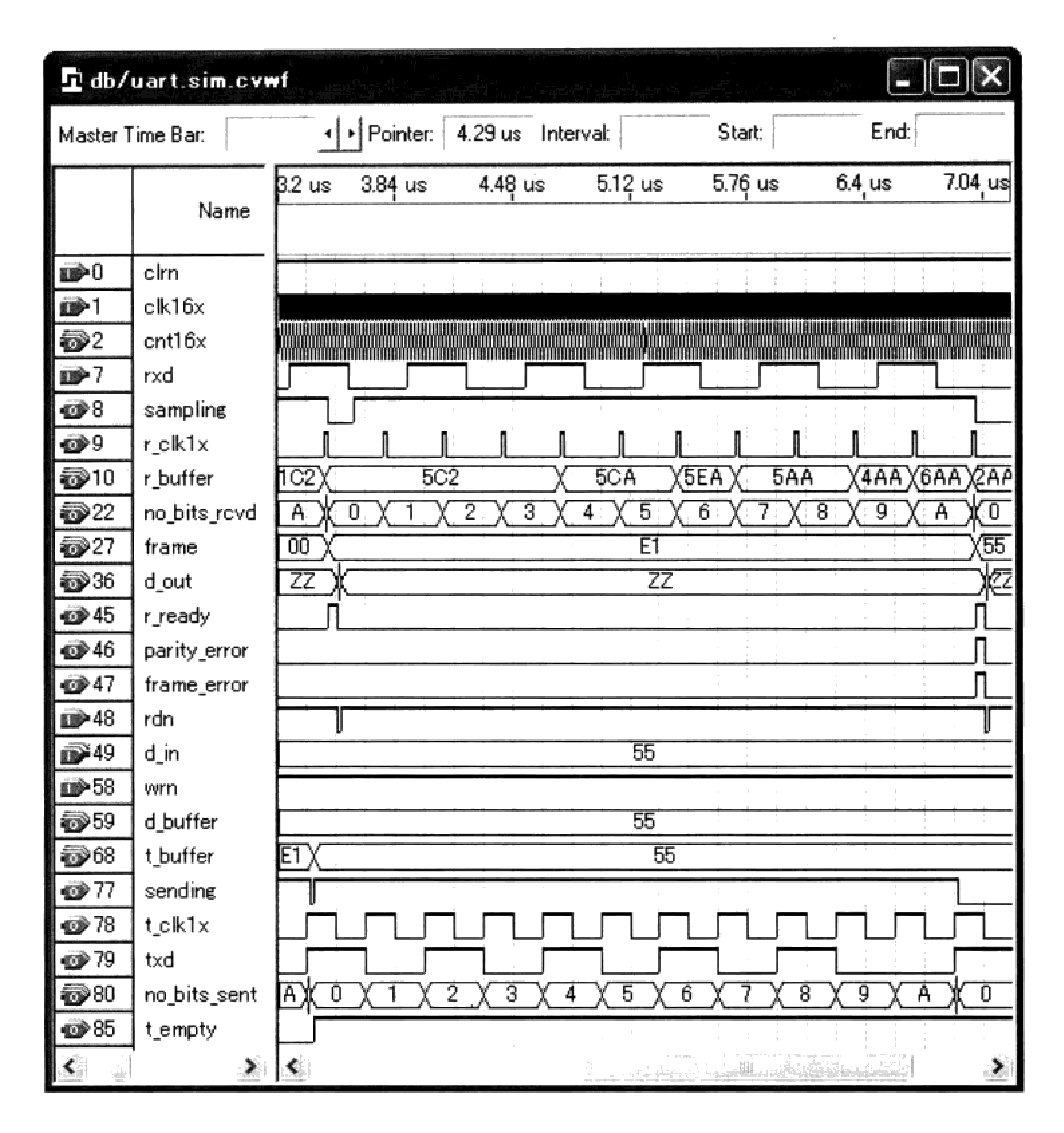


图 15.13　UART 仿真波形 (2)

图 15.12 和图 15.13 示出了 UART 的仿真波形。图中的两个计数器 no_bits_rcvd 和 no_bits_sent 的值 A 的右面还有一个 B，由于时间太短没能显示出来。图 15.12 接收和发送的数据都是 E1。图 15.13 接收和发送的数据都是 55。图 15.13 中我们故意把 rxd 的格式搞错，用以检查 parity_error 和 frame_error 两个信号。

15.4　PS/2 接口

PS/2 是个人计算机串行 I/O 端口的一种标准，名称的由来是因为它在 IBM PS/2 (Personal System/2) 机器上首次使用，连接 PS/2 键盘和 PS/2 鼠标。PS/2 端口的连接器的名称是 Mini-DIN，它有 6 个针 / 孔 (一边是针、一边是孔)，其中的两个未被使

用，其余 4 个分别是时钟、数据、VCC 和 GND。有些计算机或 FPGA 板只有一个连接器，但把两个未被使用的孔用来提供第二套的时钟和数据。这样，在外面使用一个 Y 形的分叉器，就能同时连接键盘和鼠标了。

15.4.1 PS/2 键盘

当你按下一个一般的键，键盘侧送出相应键的扫描码；松开时，送出 F0 接着又是扫描码。前者称为 Make Code，后者称为 Break Code。不一般的键，也称扩展键，在二者最前面又加上 E0。

比如，如果你按下“L”键再放开，则送出 4B F0 4B (扫描码是 4B)。

再比如，如果你按下“Delete”键再放开，则送出 E0 71 E0 F0 71 (扩展键)。

如果你长时间按下“L”键再放开，则送出 4B 4B · · · 4B F0 4B。

多个键可以同时被按下，比如先按左“Shift”(扫描码为 12)、再按“L”、放开“L”、再放开“Shift”，则送出 12 4B F0 4B F0 12。

以上的数字均是十六进制的。键盘送给主机的数据以字节为单位。发送时以串行方式进行，与 UART 的格式类似，但有同步的时钟信号，另外规定使用奇校验。时序见图 15.14，时钟高电平时键盘开始送数据，时钟低电平时主机开始读数据。

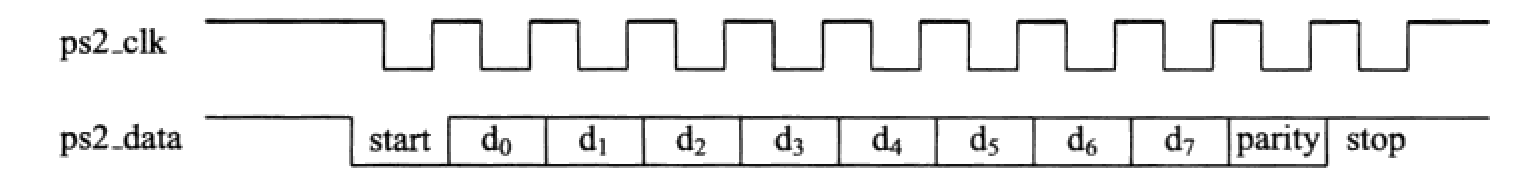


图 15.14 PS/2 键盘送数据给主机的时序图

以下的代码只负责接收由键盘送来的数据，到底是什么键交由软件处理。我们专门预备了一个 8 字节的缓冲区，以防数据丢失。首先检测时钟的下降沿，然后开始逐位接收数据并放入缓冲区。缓冲区实际上是一个先进先出的队列，配备有写指针和读指针。当队列不空时，送出 ready 信号；当队列溢出时，送出 overflow 信号。

```
module ps2_keyboard (clk, clrn, ps2_clk, ps2_data,
                     rdn, data, ready, overflow, count);
    input clk, clrn, ps2_clk, ps2_data;
    input rdn;
    output [7:0] data;
    output ready;
    output overflow;
    output [3:0] count;  // internal signal, for test
    reg overflow;        // fifo overflow
    reg [3:0] count;     // count ps2_data bits
    reg [9:0] buffer;    // ps2_data bits
    reg [7:0] fifo[7:0]; // data fifo
    reg [2:0] w_ptr, r_ptr;  // fifo write and read pointers
```

```
    // detect falling edge of ps2_clk
    reg [2:0] ps2_clk_sync;
    always @ (posedge clk) begin
        ps2_clk_sync <= {ps2_clk_sync[1:0],ps2_clk};
    end
    wire sampling = ps2_clk_sync[2] & ~ps2_clk_sync[1];
    always @ (posedge clk) begin
        if (clrn == 0) begin
            count <= 0;
            w_ptr <= 0;
            r_ptr <= 0;
            overflow <= 0;
        end else if (sampling) begin
            if (count == 4'd10) begin
                if ((buffer[0] == 0) && // start bit
                    (ps2_data)       && // stop bit
                    (^buffer[9:1])) begin // odd prity
                    fifo[w_ptr] <= buffer[8:1];  // kbd scan code
                    w_ptr <= w_ptr + 3'b1;
                    overflow <= overflow |
                                (r_ptr == (w_ptr + 3'b1));
                end
                count <= 0; // for next
            end else begin
                buffer[count] <= ps2_data; // store ps2_data
                count <= count + 3'b1;     // count ps2_data bits
            end
        end
        if (!rdn && ready) begin
            r_ptr <= r_ptr + 3'b1;
            overflow <= 0;
        end
    end
    assign ready = (w_ptr != r_ptr);
    assign data = fifo[r_ptr];
endmodule
```

图 15.15 是以上代码的仿真波形，显示的是接收“L”键的扫描码 4B 时的情形。4B (01001011_2) 的最低位先到达。图中的 count 是内部信号，拉出来是为了看着方便。由于一共要接收 11 位 (1 位起始位、8 位数据、1 位校验位和 1 位停止位)，所以计数值从 0 计到 A。当数据被读走 (rdn 为 0) 后，数据线 data 送出 0。

需要注意的是以上代码中的 ps2_clk 和 ps2_data 均定义成了输入信号，而实际上它们应该是双向的信号。主机也可以发送命令给键盘，比如复位命令 (FF) 以及表示 Num Lock、Caps Lock 和 Scroll Lock 的 LED 灯是否点亮的命令 (ED) 等。有关双向信号的电路设计将在 15.4.2 小节描述。

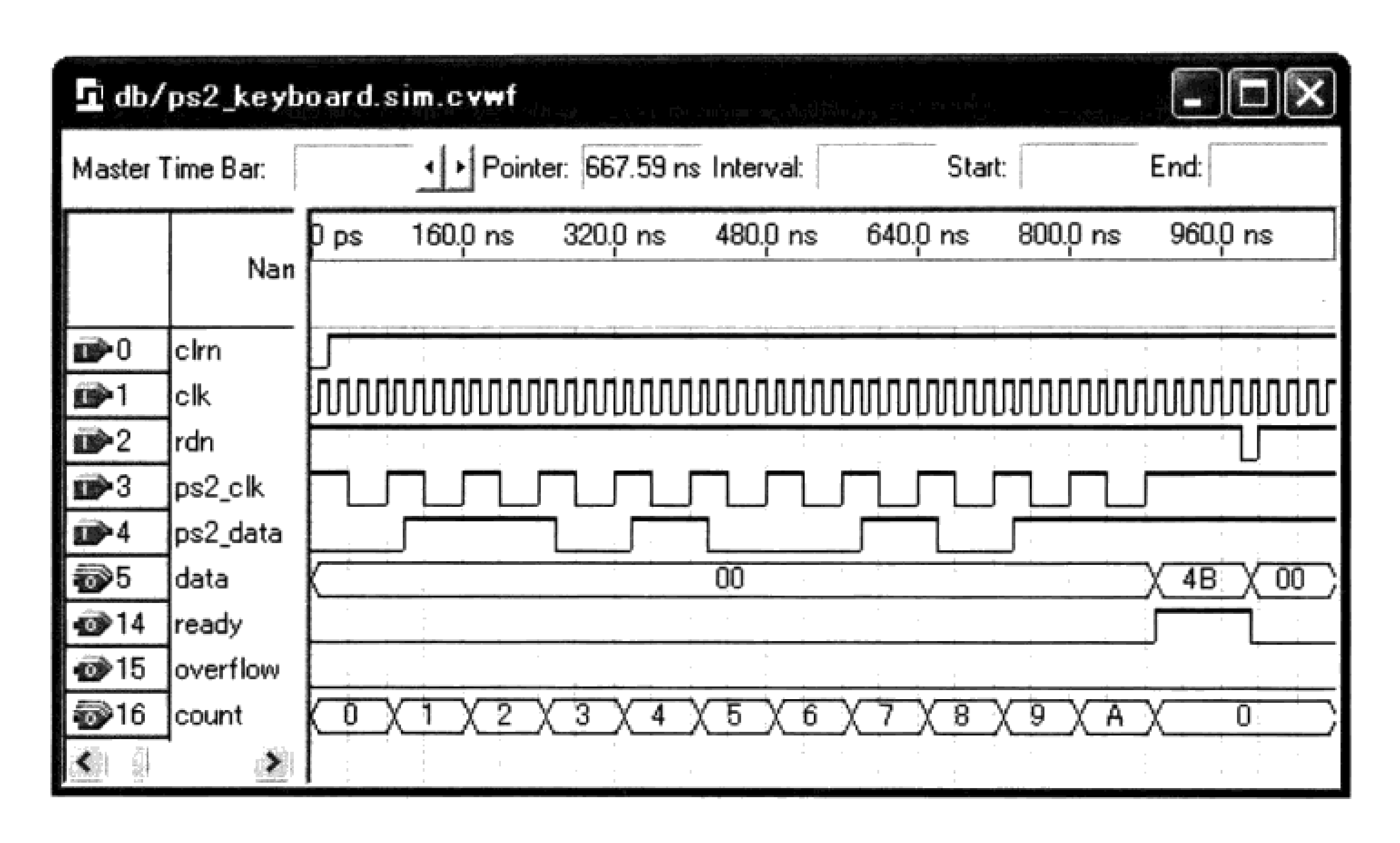


图 15.15　PS/2 键盘仿真波形

15.4.2　PS/2 鼠标

PS/2 键盘送扫描码给主机。与此类似，PS/2 鼠标送移动信息及按钮的状态信息给主机。图 15.16 给出的是三按钮鼠标 (中间按钮带一小轮的那种) 的信息格式，共有四个字节。如果是两按钮的鼠标，只有前三个字节。

	7	6	5	4	3	2	1	0
Byte 1	Y 上溢	X 上溢	Y_8	X_8	1	中按钮	右按钮	左按钮
Byte 2	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0
Byte 3	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
Byte 4	Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0

图 15.16　PS/2 鼠标数据格式

鼠标的左右移动量用 9 位补码表示的带符号数 $X_8 \cdots X_0$ 表示，鼠标向左移为负，右移为正；上下移动量用 9 位补码表示的带符号数 $Y_8 \cdots Y_0$ 表示，鼠标向下移为负，上移为正；小轮子的转动用 $Z_7 \cdots Z_0$ 表示。第一个字节中除了 X_8 和 Y_8，其他位的意义如下：X (Y) 上溢表示水平 (垂直) 移动量超出了表数范围，左、中、右按钮按下时相应的位为 1。

主机可以发送命令给鼠标来选择不同的工作方式。最常用的是所谓的流方式 (Stream Mode)：当移动鼠标或按下按钮时，鼠标立即送出连续的数据。加电后，鼠标本身进行内部测试，送出一个字节的 AA，表示已经通过了测试。然后送出 00，表示自己是一个 PS/2 鼠标。主机收到 AA 00 后，要送出 F4 (Enable)，以允许鼠标进入流方式工作状态。鼠标收到 F4 后，送出 FA 作为回答。这时鼠标处在流方式工作状

态，可以发出通常的数据包了。

注意，如果你使用 FPGA，刚加电时 FPGA 芯片还没被初始化，无法检测到 AA 00。FPGA 可以忽略这两个字节的数据，直接送出 F4 并检测 FA。FPGA 可以在任何时候送出 FF (Soft Reset) 给鼠标，令其进行初始化。鼠标收到这个命令后，送出 FA 作为回答，然后进行内部测试，再送出 AA 00 (与加电时的动作相同)。

主机与鼠标之间的信号线有两条：时钟信号 ps2_clk 和数据信号 ps2_data。它们与 PS/2 插口的连接关系见图 15.17。由于鼠标使用它们来接收命令和发送数据，因此这两条线是双向的。在鼠标控制器一侧可以使用三态门来控制主机数据的输出：当 ps2_coe (Clock Output Enable) 为 1 时，主机时钟信号 ps2_cout 经三态门送到 ps2_clk 线上，为 0 时三态门输出为高阻；当 ps2_doe (Data Output Enable) 为 1 时，主机数据信号 ps2_dout 经三态门送到 ps2_data 线上，为 0 时三态门输出为高阻。

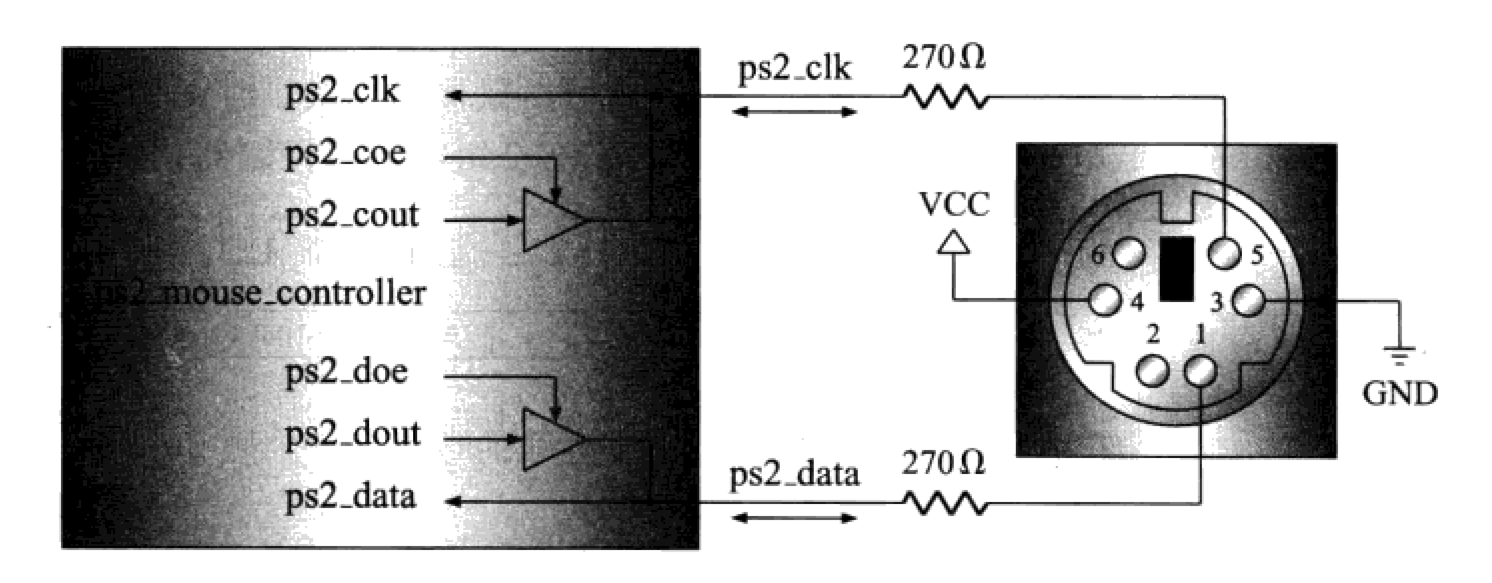


图 15.17 PS/2 鼠标控制器信号线连接关系

双向信号和三态门控制的 Verilog HDL 描述可以使用以下的例子。当使能信号为 1 时，送出二值信号；为 0 时，送出高阻信号。注意第一行的信号类型是 inout (input and output，双向)。

```
inout  ps2_clk, ps2_data; // bi-directional signals
assign ps2_clk  = ps2_coe ? ps2_cout : 1'bz;
assign ps2_data = ps2_doe ? ps2_dout : 1'bz;
```

鼠标从主机接收数据 (命令) 的时序如图 15.18 所示。图中 ps2_clk 和 ps2_data 的虚线部分由主机送出，实线部分由键盘送出。注意图中最上面的 4 个信号是鼠标控制器的内部信号。

首先，由主机送出低电平的 ps2_cout 到 ps2_clk，告诉鼠标，主机要发送数据了。然后释放 ps2_clk 并把起始位经由 ps2_dout 送至 ps2_data。鼠标检测到起始位的下降沿后，开始驱动 ps2_clk，送出一个负脉冲。主机检测到 ps2_clk 的负脉冲后，送出数据的最低位 (d_0)。然后依次送出数据的其他位，时钟仍由键盘侧提供。当奇校验位送出后，主机释放 ps2_data 线 (由提拉电阻送出停止位)。鼠标在收到停止位后，送

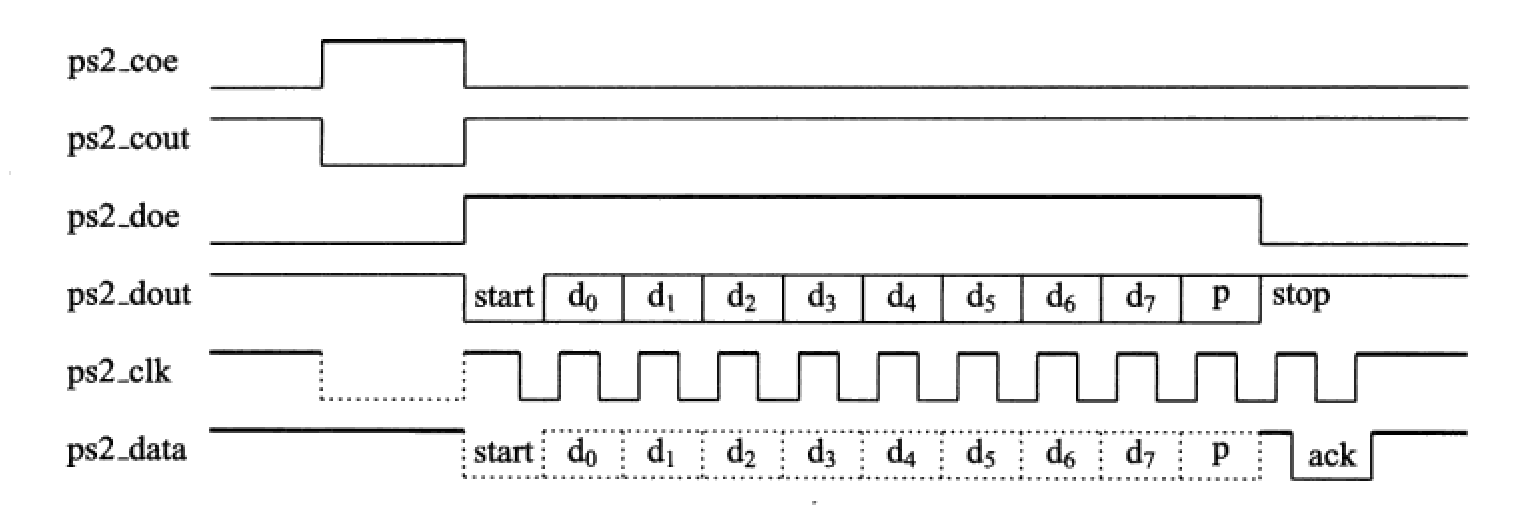


图 15.18 主机送控制字给 PS/2 鼠标的时序图

出一个低电平的 ACK 到 ps2_data 线。然后鼠标要忙着执行刚收到的命令 (比如软复位命令 FF)，并告诉主机执行结果 (比如送出 FA AA 00)。

鼠标送数据给主机的时序见图 15.19。时钟信号由鼠标侧提供。时钟高电平时鼠标开始送数据，时钟低电平时主机开始读数据，与 PS/2 键盘接口控制器相同。

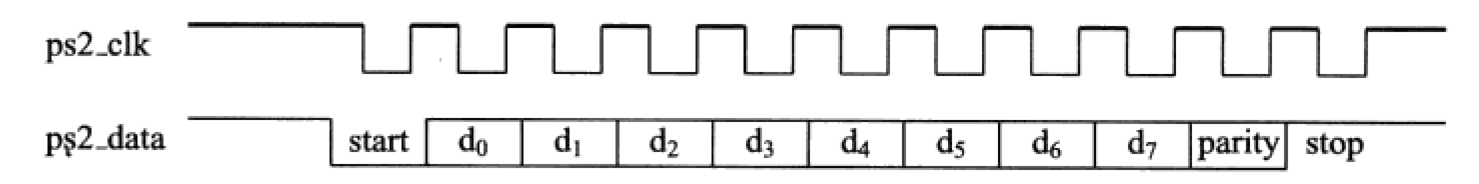


图 15.19 PS/2 鼠标送数据字给主机的时序图

读者可以根据以上的描述，设计 PS/2 鼠标控制器的 Verilog HDL 代码 (教科书作者一般总是把难题留给读者)。

15.5 视频图像阵列 VGA

本节介绍 VGA，给出它的接口控制器设计的 Verilog HDL 代码，并演示如何在 VGA 上显示键盘字符。

15.5.1 VGA 及其接口控制器设计

VGA (Video Graphics Array) 是 IBM 公司制定的一种视频数据的传输标准。它的接口信号主要有 5 个：R (Red)、G (Green)、B (Blue)、HS (Horizontal Synchronization) 和 VS (Vertical Synchronization)，即红、绿、蓝、水平同步和垂直同步。水平同步和垂直同步又称行 (Line) 同步和帧 (Frame) 同步。这些都是模拟 (Analogue) 信号，用于连接诸如 CRT 和 LCD 等显示器。以下简称红、绿、蓝为 RGB。

计算机生成的是数字 (Digital) 信号，因此需要有 D/A 转换器。液晶显示器可以直接显示数据，但为了兼容，液晶显示器也提供 VGA 接口。这就需要再把模拟信号经由 A/D 转换成数字信号。从 D 到 A 再到 D 会造成精度的损失，因此最好使用数字视频接口 DVI (Digital Visual Interface)。本节只讨论 VGA。

图像的显示是以像素 (Pixel) 为单位从左上角开始一行一行进行的。行同步信号是负脉冲。负脉冲来时要由 RGB 送出在当前行显示的像素。下一个负脉冲用来显示下一行。当整个屏幕 (一帧) 显示一遍后，由帧同步信号送出一个负脉冲，又从左上角开始显示，如图 15.20 所示。

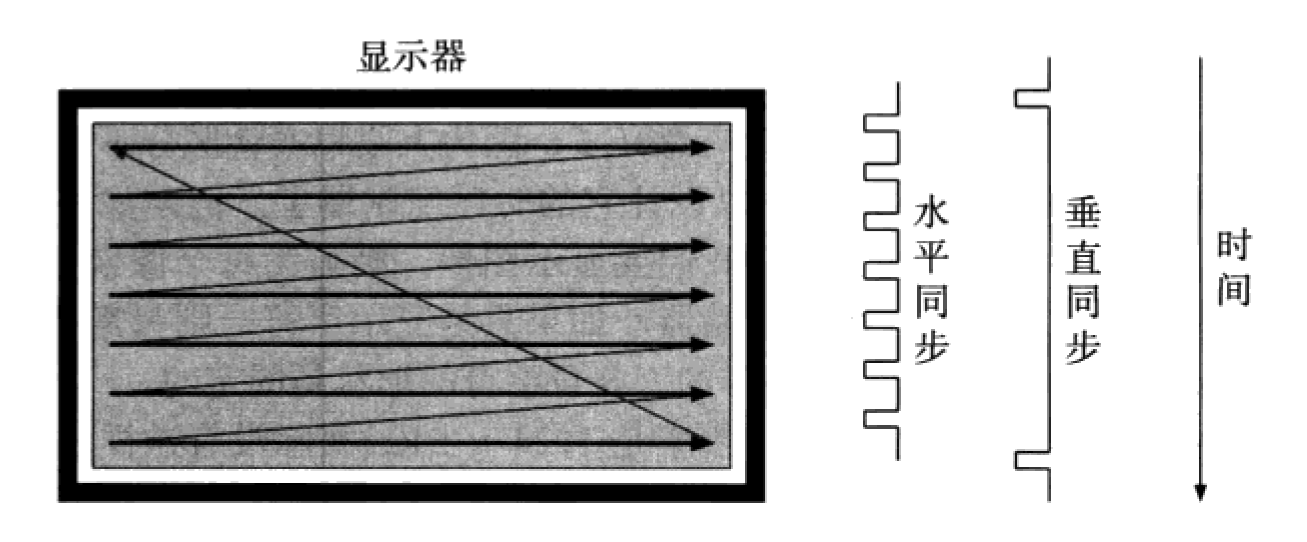


图 15.20 VGA 同步信号

显示器 (CRT 或 LCD) 的分辨率 (Resolution) 是指屏幕每行有多少个像素及每帧有多少行。标准的 VGA 的分辨率是 640 × 480，即每行有 640 个像素，每列有 480 个像素。高分辨率的显示器有 XGA (1024 × 768)、SXGA (1280 × 1024)、UXGA (1600 × 1200)、OXGA (2048 × 1536)、OSXGA (2560 × 2048) 以及宽屏的 WXGA (1366 × 768)、WSXGA (1680 × 1050) 和 WUXGA (1920 × 1200) 等。从视觉效果考虑，每秒显示的帧数不应小于 24。分辨率越高，每秒送出的像素数也应越多。由于传送速率的限制，有些特高分辨率的显示器每秒也只能刷新十几帧。这对像作者一样眼神差一些的人来讲也是完全没有问题的。

并不是所有的时间都在传送像素。由于 CRT 的电子束回扫 (从一行的尾到下一行的头或从一帧的尾到下一帧的头) 也需要时间，这时必须要让 RGB 送出电压为 0 的信号 (黑色)。图 15.21 所示的是 VGA 的同步信号长度以及何时才能送出像素。图中水平同步的数字是像素数，垂直同步的数字是行数。

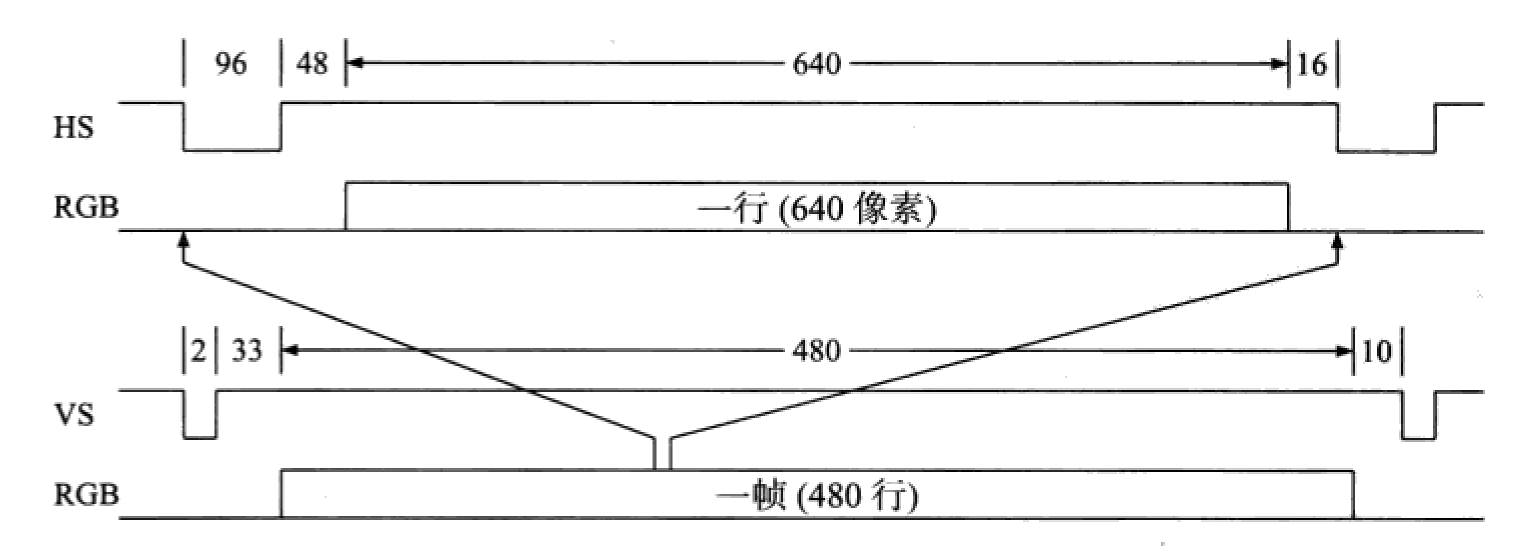


图 15.21 VGA 水平同步和垂直同步信号的长度

如果负责送出像素的时钟的频率是 25.2MHz，我们可以计算出每秒可以刷新多少帧：$25.2 \times 10^6/[(96+48+640+16)\times(2+33+480+10)] = 60$。

像素的颜色由模拟信号 RGB 的电压决定，而这三个信号是 D/A 转换器的输出，因此原始的表示红绿蓝的三个数据的位数就决定了能够显示的颜色的数量。所谓的真彩色 (True Color) 是指每个数据都有 8 位，即用 $3 \times 8 = 24$ 位来表示一个像素，其颜色的数量就有 $2^{24} = 16\,777\,216$ 种。如果你看到产品广告说有 1677 万种色彩，就是这个意思。也有用 27 位的，再多就没有意义了，因为人眼根本就区分不出来。

搞懂了 VGA 的时序，我们就可以设计它的时序控制器了。但是如何产生 RGB 相对应的数据还是有些讲究的。如果我们要显示图像，则需为每个像素指定一个颜色，这就需要有一个比较大的视频存储器。如果只是显示字符，则只需要较小的视频缓冲区，因为一个字符的颜色是一样的，并且字符的形状是固定的。

为了计算上的方便，我们假设分辨率是 1024×512 (实际没有这样的产品)。如果要用真彩色来显示图像 (比如照片)，那么视频存储器的容量需要多大呢？很简单，答案是 $512\text{K} \times 24$ 位，即 1.5MB。需要 19 位地址，每次读出 24 位。如果不要求显示图像而只是作为字符显示器来用，就不需要这么大的存储器了。假设有 128 种不同的字符，每个字符也用真彩色来显示，一个字符用 16×8 的点阵来表示 (16 行 8 列)，那么需要多大容量的存储器呢？停一下先算算看，搞清楚了就会设计电路了。

解答如下。一个字符可用 7 位来表示 ($\log_2 128 = 7$)，另由一个 24 位的数据来表示这个字符的颜色。一帧有 $512/16 = 32$ 行，每行有 $1024/8 = 128$ 个字符。因此，显示缓冲区需要有 $(128 \times 32) \times (7 + 24) = 4\text{K} \times 31$ 位，即 15.5KB。另外还要有一个字模产生器 (字模表)，它的容量是 $128 \times 16 \times 8 = 2\text{K} \times 8$ 位，即 2KB。一共需要 17.5KB，大约是 1.5MB 的 1% (注意：1M = 1024K)。如果全屏字符只用一种颜色，比如黑底白字，则每个字符的 24 位表示颜色的数据也可以省去。设计时需要注意字模点阵送出的次序：16 行像素显示一行字符。当然，如果已经有了视频存储器，则可以把字符点阵送入视频存储器。

为了演示如何用点阵表示字符，以下我们给出一个 ASCII 字符点阵的例子。假设它存放在一个文件中，文件名是 font8.c。一共有 96 个字符 (ASCII 码 0x20 - 0x7f)，每个字符用 8×8 的点阵表示，占用 8 个字节。想不想知道每个字符的形状？

```
const unsigned char Font[][8] = {
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},    /*<SPACE>  20  */
    {0x18,0x18,0x18,0x18,0x00,0x18,0x18,0x00},    /*   !     21  */
    {0x6c,0x6c,0x48,0x00,0x00,0x00,0x00,0x00},    /*   "     22  */
    {0x6c,0x6c,0xfe,0x6c,0xfe,0x6c,0x6c,0x00},    /*   #     23  */
    {0x18,0x7e,0xd8,0x7e,0x1b,0x7e,0x18,0x00},    /*   $     24  */
    {0x62,0x66,0x0c,0x18,0x30,0x66,0x46,0x00},    /*   %     25  */
    {0x38,0x6c,0x68,0x76,0xdc,0xcc,0x76,0x00},    /*   &     26  */
    {0x18,0x18,0x30,0x00,0x00,0x00,0x00,0x00},    /*   '     27  */
    {0x0c,0x18,0x30,0x30,0x30,0x18,0x0c,0x00},    /*   (     28  */
    {0x30,0x18,0x0c,0x0c,0x0c,0x18,0x30,0x00},    /*   )     29  */
    {0x00,0x6c,0x38,0xfe,0x38,0x6c,0x00,0x00},    /*   *     2a  */
```

```
{0x00,0x18,0x18,0x7e,0x18,0x18,0x00,0x00},    /*  +    2b  */
{0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x10},    /*  ,    2c  */
{0x00,0x00,0x00,0x7e,0x00,0x00,0x00,0x00},    /*  -    2d  */
{0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x00},    /*  .    2e  */
{0x02,0x06,0x0c,0x18,0x30,0x60,0x40,0x00},    /*  /    2f  */
{0x3c,0x66,0x6e,0x76,0x66,0x66,0x3c,0x00},    /*  0    30  */
{0x18,0x18,0x38,0x18,0x18,0x18,0x3c,0x00},    /*  1    31  */
{0x7c,0x06,0x06,0x3c,0x60,0x60,0x7c,0x00},    /*  2    32  */
{0x7c,0x06,0x06,0x3c,0x06,0x06,0x7c,0x00},    /*  3    33  */
{0x66,0x66,0x66,0x7e,0x06,0x06,0x06,0x00},    /*  4    34  */
{0x7e,0x60,0x60,0x7c,0x06,0x06,0x7c,0x00},    /*  5    35  */
{0x3c,0x60,0x60,0x7c,0x66,0x66,0x3c,0x00},    /*  6    36  */
{0x7e,0x06,0x0c,0x18,0x18,0x18,0x18,0x00},    /*  7    37  */
{0x3c,0x66,0x66,0x3c,0x66,0x66,0x3c,0x00},    /*  8    38  */
{0x3c,0x66,0x66,0x3e,0x06,0x06,0x3c,0x00},    /*  9    39  */
{0x00,0x18,0x18,0x00,0x18,0x18,0x00,0x00},    /*  :    3a  */
{0x00,0x00,0x18,0x18,0x00,0x18,0x18,0x10},    /*  ;    3b  */
{0x0c,0x18,0x30,0x60,0x30,0x18,0x0c,0x00},    /*  <    3c  */
{0x00,0x00,0x7e,0x00,0x7e,0x00,0x00,0x00},    /*  =    3d  */
{0x30,0x18,0x0c,0x06,0x0c,0x18,0x30,0x00},    /*  >    3e  */
{0x3c,0x66,0x06,0x1c,0x18,0x00,0x18,0x00},    /*  ?    3f  */
{0x3c,0x66,0x6e,0x6a,0x6e,0x60,0x3e,0x00},    /*  @    40  */
{0x3c,0x66,0x66,0x7e,0x66,0x66,0x66,0x00},    /*  A    41  */
{0x7c,0x66,0x66,0x7c,0x66,0x66,0x7c,0x00},    /*  B    42  */
{0x3c,0x66,0x60,0x60,0x60,0x66,0x3c,0x00},    /*  C    43  */
{0x7c,0x66,0x66,0x66,0x66,0x66,0x7c,0x00},    /*  D    44  */
{0x7e,0x60,0x60,0x7c,0x60,0x60,0x7e,0x00},    /*  E    45  */
{0x7e,0x60,0x60,0x7c,0x60,0x60,0x60,0x00},    /*  F    46  */
{0x3c,0x66,0x60,0x6e,0x66,0x66,0x3c,0x00},    /*  G    47  */
{0x66,0x66,0x66,0x7e,0x66,0x66,0x66,0x00},    /*  H    48  */
{0x3c,0x18,0x18,0x18,0x18,0x18,0x3c,0x00},    /*  I    49  */
{0x3e,0x0c,0x0c,0x0c,0x0c,0x6c,0x38,0x00},    /*  J    4a  */
{0x66,0x6c,0x78,0x70,0x78,0x6c,0x66,0x00},    /*  K    4b  */
{0x60,0x60,0x60,0x60,0x60,0x60,0x7e,0x00},    /*  L    4c  */
{0xc6,0xee,0xfe,0xd6,0xc6,0xc6,0xc6,0x00},    /*  M    4d  */
{0x66,0x66,0x76,0x7e,0x6e,0x66,0x66,0x00},    /*  N    4e  */
{0x3c,0x66,0x66,0x66,0x66,0x66,0x3c,0x00},    /*  O    4f  */
{0x7c,0x66,0x66,0x7c,0x60,0x60,0x60,0x00},    /*  P    50  */
{0x3c,0x66,0x66,0x66,0x6e,0x66,0x3e,0x00},    /*  Q    51  */
{0x7c,0x66,0x66,0x7c,0x66,0x66,0x66,0x00},    /*  R    52  */
{0x3e,0x60,0x60,0x3c,0x06,0x06,0x7c,0x00},    /*  S    53  */
{0x7e,0x18,0x18,0x18,0x18,0x18,0x18,0x00},    /*  T    54  */
{0x66,0x66,0x66,0x66,0x66,0x66,0x3c,0x00},    /*  U    55  */
{0x66,0x66,0x66,0x66,0x3c,0x3c,0x18,0x00},    /*  V    56  */
{0xc6,0xc6,0xd6,0xd6,0xfe,0xee,0x44,0x00},    /*  W    57  */
```

```
    {0x66,0x66,0x3c,0x18,0x3c,0x66,0x66,0x00},    /*   X     58  */
    {0x66,0x66,0x66,0x3c,0x18,0x18,0x18,0x00},    /*   Y     59  */
    {0x7e,0x06,0x0c,0x18,0x30,0x60,0x7e,0x00},    /*   Z     5a  */
    {0x3c,0x30,0x30,0x30,0x30,0x30,0x3c,0x00},    /*   [     5b  */
    {0x40,0x60,0x30,0x18,0x0c,0x06,0x02,0x00},    /*   \     5c  */
    {0x3c,0x0c,0x0c,0x0c,0x0c,0x0c,0x3c,0x00},    /*   ]     5d  */
    {0x10,0x38,0x6c,0x00,0x00,0x00,0x00,0x00},    /*   ^     5e  */
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff},    /*   _     5f  */
    {0x18,0x18,0x0c,0x00,0x00,0x00,0x00,0x00},    /*   `     60  */
    {0x00,0x00,0x3c,0x06,0x3e,0x66,0x3a,0x00},    /*   a     61  */
    {0x60,0x60,0x7c,0x66,0x66,0x66,0x7c,0x00},    /*   b     62  */
    {0x00,0x00,0x3c,0x66,0x60,0x66,0x3c,0x00},    /*   c     63  */
    {0x06,0x06,0x3e,0x66,0x66,0x66,0x3e,0x00},    /*   d     64  */
    {0x00,0x00,0x3c,0x66,0x7c,0x60,0x3c,0x00},    /*   e     65  */
    {0x0e,0x18,0x18,0x3e,0x18,0x18,0x18,0x00},    /*   f     66  */
    {0x00,0x00,0x3e,0x66,0x66,0x3e,0x06,0x3c},    /*   g     67  */
    {0x60,0x60,0x7c,0x66,0x66,0x66,0x66,0x00},    /*   h     68  */
    {0x18,0x00,0x18,0x18,0x18,0x18,0x18,0x00},    /*   i     69  */
    {0x18,0x00,0x18,0x18,0x18,0x18,0x18,0x70},    /*   j     6a  */
    {0x60,0x60,0x66,0x6c,0x78,0x6c,0x66,0x00},    /*   k     6b  */
    {0x30,0x30,0x30,0x30,0x30,0x30,0x1c,0x00},    /*   l     6c  */
    {0x00,0x00,0xcc,0xfe,0xd6,0xc6,0xc6,0x00},    /*   m     6d  */
    {0x00,0x00,0x7c,0x66,0x66,0x66,0x66,0x00},    /*   n     6e  */
    {0x00,0x00,0x3c,0x66,0x66,0x66,0x3c,0x00},    /*   o     6f  */
    {0x00,0x00,0x7c,0x66,0x66,0x7c,0x60,0x60},    /*   p     70  */
    {0x00,0x00,0x3e,0x66,0x66,0x3e,0x06,0x06},    /*   q     71  */
    {0x00,0x00,0x36,0x38,0x30,0x30,0x30,0x00},    /*   r     72  */
    {0x00,0x00,0x3e,0x60,0x3c,0x06,0x7c,0x00},    /*   s     73  */
    {0x18,0x18,0x3c,0x18,0x18,0x18,0x0c,0x00},    /*   t     74  */
    {0x00,0x00,0x66,0x66,0x66,0x66,0x3c,0x00},    /*   u     75  */
    {0x00,0x00,0x66,0x66,0x66,0x3c,0x18,0x00},    /*   v     76  */
    {0x00,0x00,0xc6,0xd6,0xd6,0x7c,0x28,0x00},    /*   w     77  */
    {0x00,0x00,0x66,0x3c,0x18,0x3c,0x66,0x00},    /*   x     78  */
    {0x00,0x00,0x66,0x66,0x66,0x3e,0x06,0x7c},    /*   y     79  */
    {0x00,0x00,0x7e,0x0c,0x18,0x30,0x7e,0x00},    /*   z     7a  */
    {0x1c,0x30,0x30,0x60,0x30,0x30,0x1c,0x00},    /*   {     7b  */
    {0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x00},    /*   |     7c  */
    {0x38,0x0c,0x0c,0x06,0x0c,0x0c,0x38,0x00},    /*   }     7d  */
    {0x00,0x32,0x4c,0x00,0x00,0x00,0x00,0x00},    /*   ~     7e  */
    {0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}     /* <DEL>   7f  */
};
```

以下的程序显示 font8.c 中的所有字符。chars_per_line 是每行显示的字符个数。如果是标准的 VGA，其值为 $640/8 = 80$。我们在程序中将其设为 16，是基于以下两点考虑的。(1) 点阵中的一位对应一个像素，但我们在程序中用一个字符来显示：

为 1 时显示大写字符“O”；为 0 时显示空白字符；(2) 本书版面宽度的限制。

```
extern unsigned char Font[][8];
main() {
    int chars_per_line = 16; // 80 x 60 characters for VGA
    int char_no;
    unsigned char char_row_bitmap;
    int row, col;
    int i;
    for (char_no = 0; char_no < 96; char_no += chars_per_line) {
        for (row = 0; row < 8; row++) {
            for (i = 0; i < chars_per_line; i++) {
                if ((char_no + i) < 96) {
                    char_row_bitmap = Font[char_no + i][row];
                    for (col = 7; col >= 0; col--) {
                        if (((char_row_bitmap >> col) & 1) == 1) {
                            printf ("O");
                        } else {
                            printf (" ");
                        }
                    }
                }
            }
            printf ("\n"); // next row
        }
    }
}
```

程序的运行结果见图 15.22，排版时把行间距调小了。最后一个字符 (Delete) 为什么要设计成全白？答案是留给读者在书写俄罗斯方块游戏程序时使用。

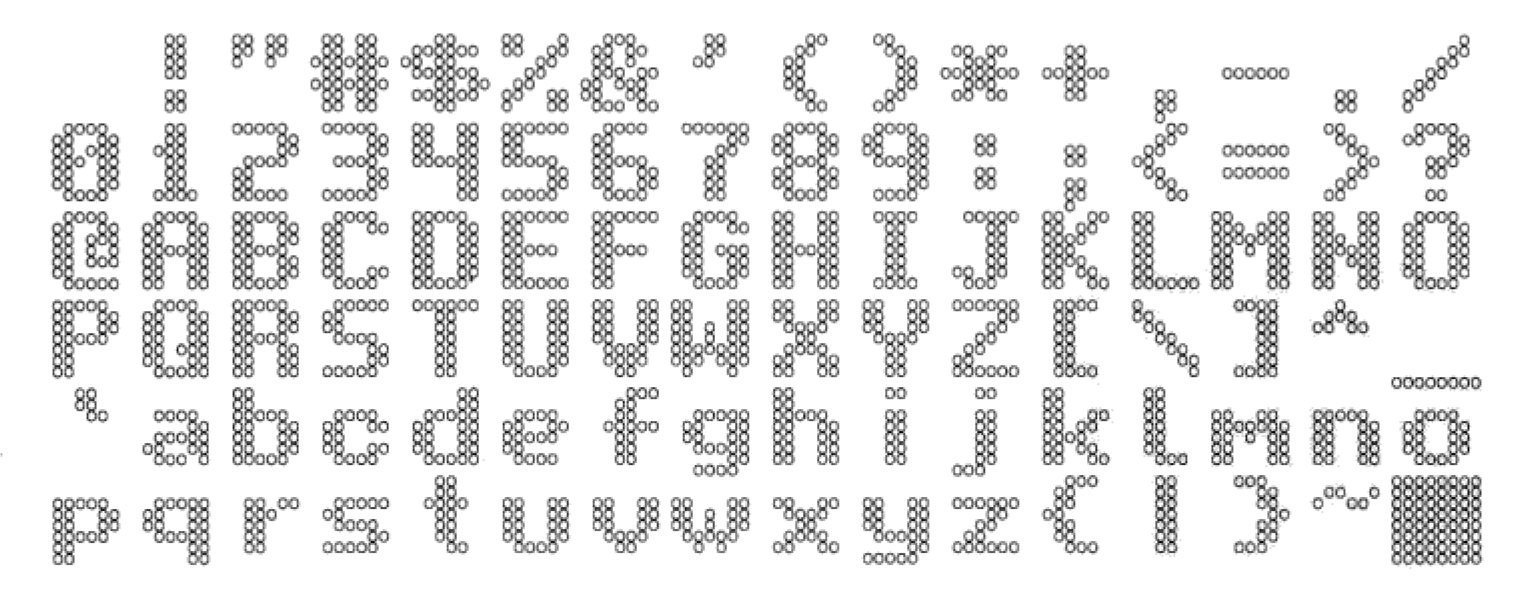


图 15.22 ASCII 字符点阵

以下描述一种简单的用于图像显示的 VGA 的时序控制器，见图 15.23。VGA 模块产生视频存储器 Pixel_RAM 的读地址，从视频存储器中读出 12 位的像素数据

(每种颜色用 4 位表示，共可表示 4096 种颜色)。读来的数据 d_in[11:0] 经锁存后由 r[3:0]、g[3:0] 和 b[3:0] 送出。D/A 转换器由简单的电阻阵列实现。注意接到每种颜色高位数据的电阻应有较低的阻抗，然后依次递增，接到最低位的电阻的阻抗值最高。阻抗值的设定应使数字信号为最大值时输出的电压为 0.7V，最小值 (0) 时输出的电压为 0V。作为参考，四个电阻的阻抗值可分别设为 510Ω、1kΩ、2kΩ 和 4kΩ。水平同步信号 hs 和垂直同步信号 vs 也经电阻 (比如 82.5Ω) 连到显示器接口。

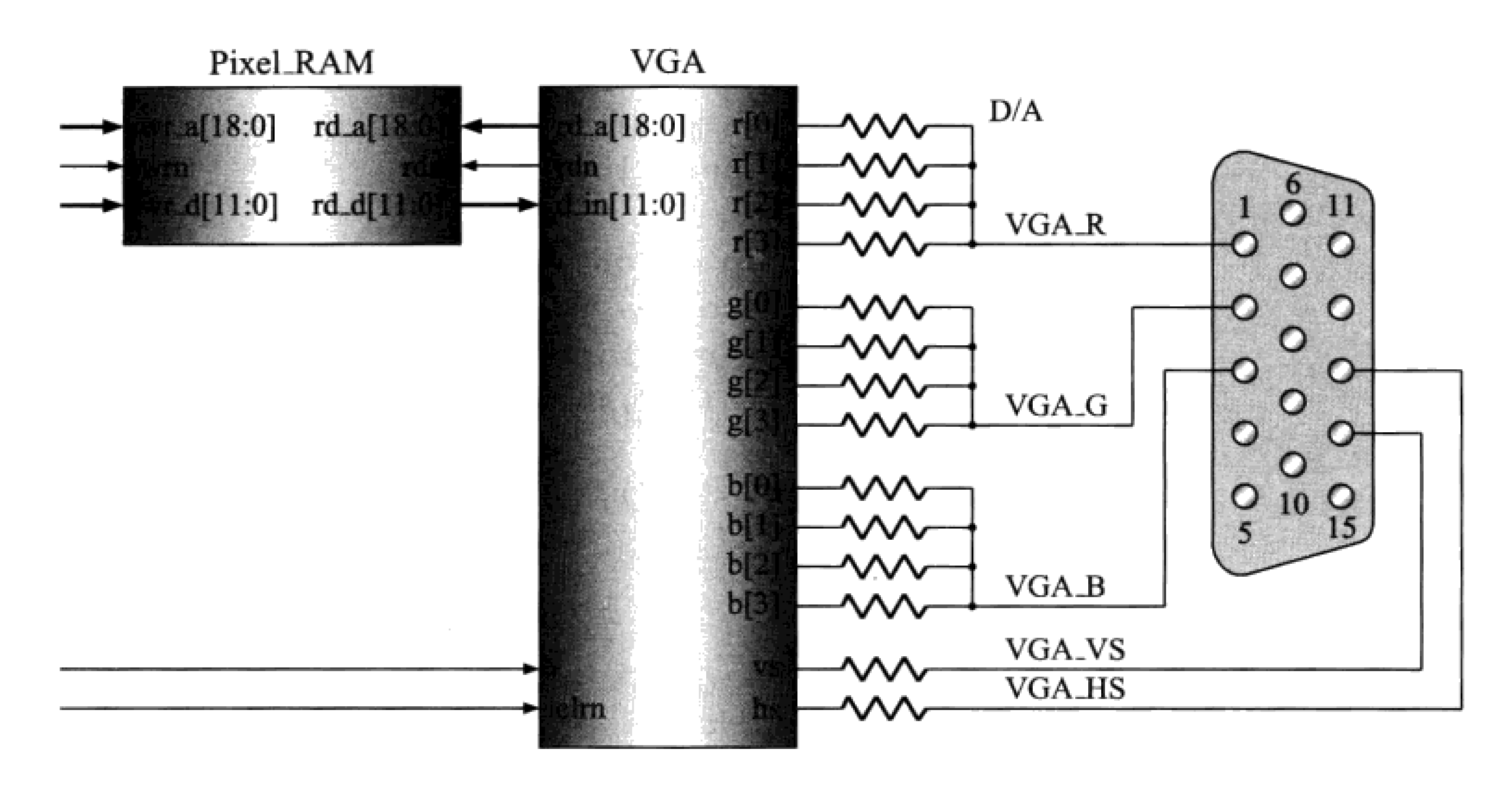


图 15.23　VGA 时序控制器

VGA 的分辨率是 640 × 480，为了简化设计及提高读地址产生的速度，我们只是把行列的像素地址拼接在一起。行地址 (row) 记录行号，因为一帧有 480 行，因此需要 9 位地址；列地址 (col) 记录列号，因为一帧有 640 列，因此需要 10 位地址。总的地址位数为 19。注意，这种做法会导致视频存储器的浪费，因为 19 位地址的存储器可以存放 1024 × 512 个像素。另外的一种产生地址的做法是行地址乘以 640 再加上列地址，但由于需要一个乘法器，导致电路成本的增加及地址产生的速度变慢。

以下是 VGA 时序控制器的 Verilog HDL 代码。输入时钟信号 clk 为 50MHz，经二分频后的时钟信号 vga_clk 为 25MHz；clrn 是低电平有效的复位信号；d_in 是从视频存储器读来的 12 位像素数据；rd_a 是视频存储器的 19 位地址；rdn 是低电平有效的读信号；r、g、b 分别是 4 位的红、绿、蓝信号；hs 和 vs 分别是水平同步和垂直同步信号；h_count 和 v_count 是内部信号，送出来仅供测试用，它们分别是水平方向和垂直方向的计数器。阅读代码时请参看图 15.21 所示的时序。

```
// VGA signal generator, 640 x 480, 25MHz
module vga (clk,clrn,d_in,rd_a,rdn,r,g,b,hs,vs,h_count,v_count);
    input  clk; // 50MHz
    input  clrn;
```

```
input  [11:0] d_in; // rrrr_gggg_bbbb, pixel
output [18:0] rd_a; // pixel RAM addr, 640 (1024) x 480 (512)
output rdn; // read pixel RAM (active_low)
output [3:0] r, g, b; // red, green, blue, 4-bit for each
output hs, vs; // horizontal and vertical synchronization
// for test
output [9:0] h_count; // VGA horizontal counter (0-799)
output [9:0] v_count; // VGA vertical counter (0-524)
// internal signals
reg vga_clk; // 25MHz
reg [9:0] h_count; // VGA horizontal counter (0-799): pixel
// __    ____________________________    ________________
//    |__|                              |__|
//    |96|    ______________________    |96|    _________
// ________|       1 line        |________| (next line)
//    |96|48|-------- 640 ---------|16|96|48|
//    | 144 |                     |  |
//    |------------ 784 -----------|  |
//    |-------------- 800 ------------|

reg [9:0] v_count; // VGA vertical counter (0-524): lines
// __    ____________________________    ________________
//    |__|                              |__|
//    | 2|    ______________________    | 2|    _________
// ________|       1 frame       |________| (next frame)
//    | 2|33|-------- 480 ---------|10| 2|33|
//    |  35 |                     |  |
//    |------------ 515 -----------|  |
//    |-------------- 525 ------------|
reg [11:0] data_reg; // latched rrrr_gggg_bbbb, pixel
reg video_out; // VGA signal enable
// vga_clk: 25MHz
always @ (posedge clk or negedge clrn) begin
    if (clrn == 0) begin
        vga_clk <= 1'b1;
    end else begin
        vga_clk <= ~vga_clk;
    end
end
// h_count: VGA horizontal counter (0-799)
always @ (posedge vga_clk or negedge clrn) begin
    if (clrn == 0) begin
        h_count <= 10'h0;
    end else if (h_count == 10'd799) begin
        h_count <= 10'h0;
```

```
        end else begin
            h_count <= h_count + 10'h1;
        end
    end
    // v_count: VGA vertical counter (0-524)
    always @ (posedge vga_clk or negedge clrn) begin
        if (clrn == 0) begin
            v_count <= 10'h0;
        end else if (h_count == 10'd799) begin
            if (v_count == 10'd524) begin
                v_count <= 10'h0;
            end else begin
                v_count <= v_count + 10'h1;
            end
        end
    end
    // video_out: VGA signal enable
    always @ (posedge vga_clk or negedge clrn) begin
        if (clrn == 0) begin
             video_out <=  1'b0;
             data_reg  <= 12'h0;
        end else begin
             video_out <= ~rdn;
             data_reg  <= d_in; // pixel RAM access time: < 40ns
        end
    end
    // rdn: one vga_clk cycle ahead due to video_out delay
    assign rdn = ~(((h_count>=10'd143) && (h_count<10'd783)) &&
                   ((v_count>=10'd35)  && (v_count<10'd515)));
    wire [9:0] row = v_count - 10'd35;
    wire [9:0] col = h_count - 10'd143;
    assign rd_a = {row[8:0],col};
    assign hs = (h_count >= 10'd96); // horizontal synchronization
    assign vs = (v_count >= 10'd2);  // vertical synchronization
    assign r = (video_out) ? data_reg[11:8] : 4'h0; // 4-bit red
    assign g = (video_out) ? data_reg[07:4] : 4'h0; // 4-bit green
    assign b = (video_out) ? data_reg[03:0] : 4'h0; // 4-bit blue
endmodule
```

由于读视频存储器需要时间，因此我们先读出像素，将其锁存后再显示。如果边读边显示的话，可能会造成屏幕显示不稳定。但如果视频存储器本身就是同步的(输出用时钟锁存)，则没必要再锁存了。本例中使用 vga_clk (25MHz) 的一个时钟周期读视频存储器，因此视频存储器的访问周期时间不应大于 40ns。

图 15.24 和图 15.25 示出的是 VGA 时序控制器的仿真波形。计数值用十进制表示。图 15.24 是送出第一个像素前后的时序。此时 v_count 等于 35 (0 ～ 34 没有像素

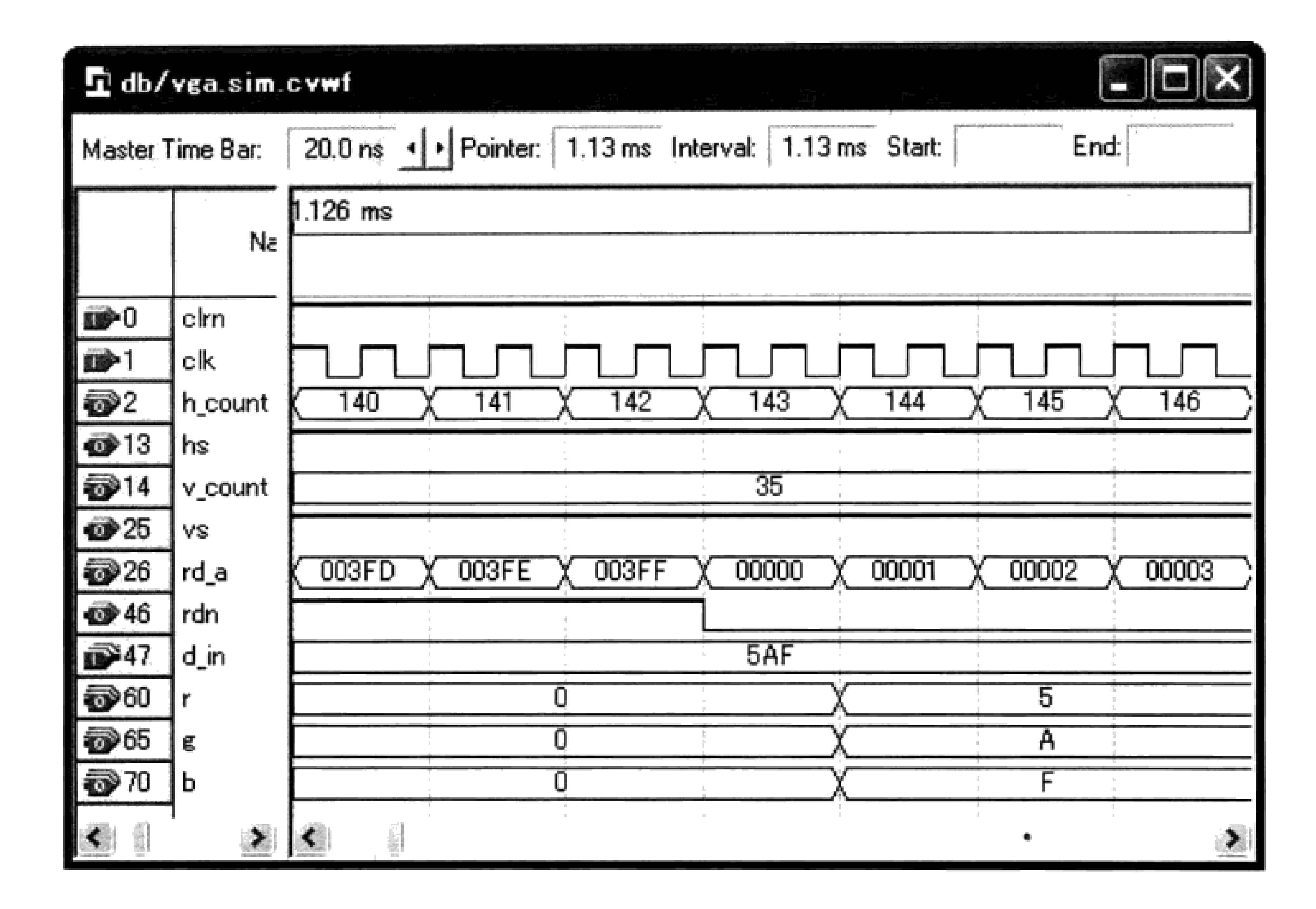


图 15.24　VGA 仿真波形 (1)

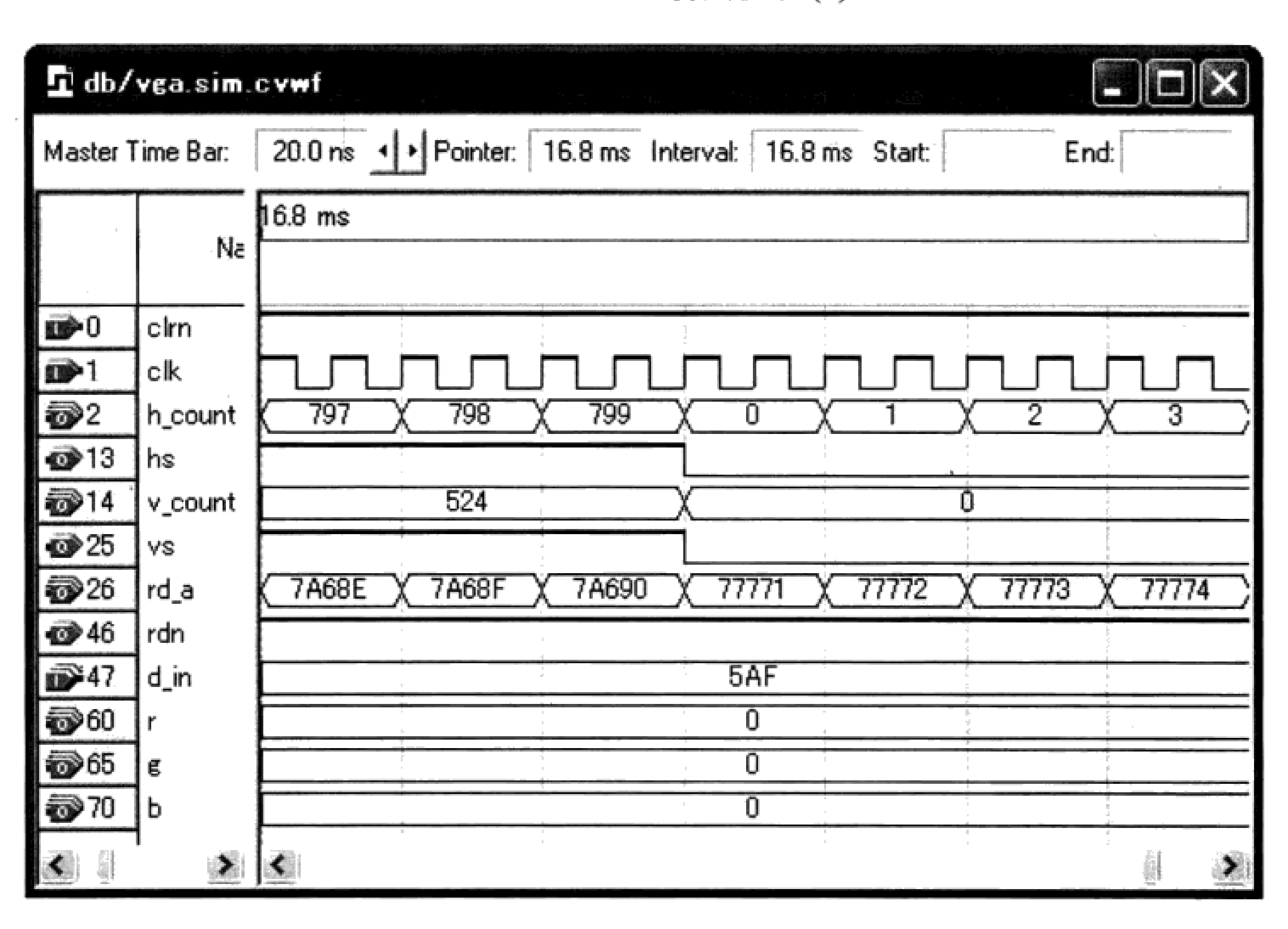


图 15.25　VGA 仿真波形 (2)

送出)，在 h_count 等于 144 处送出 rgb 数据 (0 ～ 143 没有像素送出)，而读视频存储器在它的前一个时钟周期进行 (视频存储器地址 rd_a 为 0000、低电平有效的读信号 rdn 为 0)。由于我们没有把视频存储器加进来，只是在 d_in 端输入十六进制的 5AF，因此 r、g、b 的输出总是 5、A、F。图 15.25 是下一帧垂直同步信号开始处前后的时序。在 vs 变 0 之前，v_count 已到达最大值 524，h_count 也到达最大值 799。

15.5.2　VGA 显示键盘字符

本小节给出一个具体的例子说明如何从 PS/2 键盘接收字符并在 VGA 上显示。图 15.26 示出的是该例的总体结构图。如 15.4.1 小节所述，PS/2 键盘送出的信号有两个：ps2_clk 和 ps2_data。送到 VGA 的信号有 r、g、b、hs 和 vs。由于键盘侧送出的是 Make Code 和 Break Code 字节序列，我们首先从中提取扫描码，把它转换成 ASCII 码 (7 位)，然后把 ASCII 码送入一个字符显示缓冲区。我们使用本节给出的 8 × 8 字模来表示一个字符，因此缓冲区的容量应为 80 × 60 × 7 位。为了设计方便起见，我们令缓冲区的容量为 128 × 64 × 7 位，地址为 7 + 6 = 13 位。

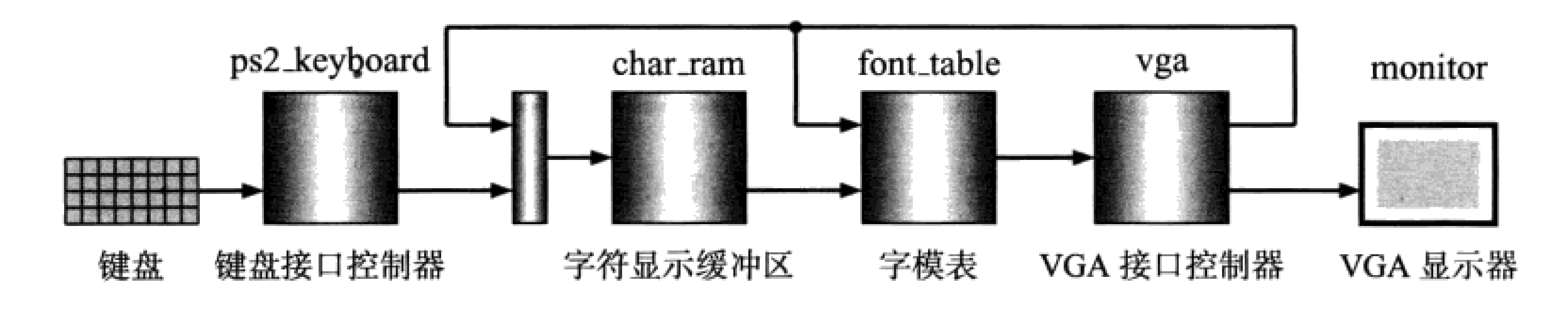


图 15.26　VGA 显示键盘字符的总体结构图

在该例中，我们没有使用视频存储器。因此从 VGA 接口控制器送出的视频存储器地址要被转换成字符显示缓冲区的地址，从中得到相应位置的字符的 ASCII 码，然后使用该 ASCII 码去访问字模表，得到点阵信息，送给 VGA 接口控制器。由于本例只用黑白两色显示字符，一个像素只用一位表示，因此 VGA 接口控制器的 12 位数据均接同一信号。它的 Verilog HDL 代码如下。

```
module vga_keyboard (clk, clrn, ps2_clk, ps2_data, r, g, b, hs, vs);
    input  clk; // 50MHz
    input  clrn;
    input  ps2_clk, ps2_data; // keyboard clock and data
    output [3:0] r, g, b; // red, green, blue, 4-bit for each
    output hs, vs; // horizontal and vertical synchronization
```

以下代码调用 vga 模块 (VGA 接口控制器)。vga_row 和 vga_col 分别是 VGA 像素的行地址和列地址；row 和 col 分别是一个 8 × 8 字模的行地址和列地址；char_row 和 char_col 分别是字符显示缓冲区的行地址和列地址；kbd_row 和 kbd_col 分别是当键盘字符要写入字符显示缓冲区时的行地址和列地址。

```
// vga
wire [18:0] rd_a;   // pixel RAM addr, 640 (1024) x 480 (512)
wire vga_rdn;       // assign rd_a = {row[8:0],col};
wire [9:0] h_count; // VGA horizontal counter (0-799)
wire [9:0] v_count; // VGA vertical counter (0-524)
vga vga1 (clk, clrn, {12{ft_do}}, rd_a, vga_rdn, r, g, b,
          hs, vs, h_count, v_count);
wire [8:0] vga_row  = rd_a[18:10];
wire [9:0] vga_col  = rd_a[9:0];
wire [2:0] row      = vga_row[2:0];
wire [2:0] col      = vga_col[2:0];
wire [5:0] char_row = cram_w ? kbd_row : vga_row[8:3];
wire [6:0] char_col = cram_w ? kbd_col : vga_col[9:3];
```

以下代码调用 char_ram 模块 (字符显示缓冲区)。cram_a 是字符显示缓冲区的地址；cram_do 是缓冲区的输出 (ASCII 码)；cram_di 是缓冲区的输入，也是 ASCII 码，由键盘扫描码转换后得到；cram_w 是缓冲区的写使能信号。char_ram 模块的 Verilog HDL 代码稍后给出。

```
// char_ram, 80 (128) x 60 (64) = 4800 (8192) chars
wire [12:0] cram_a  = {char_row,char_col};
wire [6:0] cram_do; // ascii, to font_table
char_ram charram (cram_do, cram_di, cram_a, cram_w, clk);
```

以下代码调用 font_table 模块 (字模表)。fta 是字模表的地址，由 ASCII 码、字模行地址和字模列地址拼接得到；ft_do 是字模表的输出 (像素)，只有 1 位。以上两个信号用来读 ROM 本应足够了，但 Altera 有些器件不支持异步存储器，要求加一个时钟信号进来捣乱。font_table 模块的 Verilog HDL 代码稍后给出。

```
// font_table 128 * 8 * 8 * 1
wire [12:0] fta = {cram_do,row,col};
wire ft_do; // mono color, to vga
font_table ft (fta, ~clk, ft_do); // synchronous ROM
```

以下代码调用 ps2_keyboard 模块 (键盘接口控制器)。ready 表示键盘控制器有一个字节准备好了；data 就是那个字节。

```
// ps2_keyboard
wire [3:0] count; // internal signal, for test
wire ready, overflow;
wire [7:0] data; // kbd code
ps2_keyboard kbd (clk, clrn, ps2_clk, ps2_data, kbd_rdn, data,
                  ready, overflow, count);
```

以下代码检查键盘送来的字节，看看它是否是 Shift 键，是否是 Break 代码。应该检查的项目有很多，比如是否是扩展键等，我们在这里省略了。

```
// key type
reg is_shift;
reg is_break;
always @* begin
    is_shift = 0;
    is_break = 0;
    if (ready) begin
        if (data == 8'hf0) begin // break
            is_break = 1;
        end
        if ((data == 8'h12) || (data == 8'h59)) begin // shift
            is_shift = 1;
        end
    end
end
```

以下代码定义了 4 个状态。状态 0 是非常一般的状态，准备接收一个键盘扫描码 (当按住一个键不放时，可以连续接收)。但如果是 Shift 键，把 shift_pressed 置 1；如果是 Break Code，将进入状态 1。状态 1 和状态 2 都不干什么事情，只是等待最后一个字节的到来。状态 3 是一次键盘输入的最后一个状态。如果最后一个字节是 Shift，释放 shift_pressed。最后一个状态的下一状态当然是状态 0 了。

```
// state 0, 1, 2, 3
reg [1:0] state;
reg shift_pressed;
reg key_released;
always @ (negedge clk or negedge clrn) begin
    if (clrn == 0) begin
        state <= 0;
    end else begin
        case (state)
            2'd0: begin // for make code
                if (is_shift) shift_pressed <= 1;
                if (is_break) state <= 1; end
            2'd1: begin // for break code
                if (!ready) state <= 2;   end
            2'd2: begin // waiting for last code
                if (ready) state <= 3;    end
            2'd3: begin // ignore last code
                if (is_shift) shift_pressed <= 0;
                if (!ready) state <= 0;   end
        endcase
    end
end
```

以下代码的主要任务是把键盘扫描码转换成 ASCII 码 (调用 s2a function) 并把它

写入字符显示缓冲区 (cram_w = 1)。我们在这里只转换了部分扫描码。function 中的 default 应把 0 赋给 s2a，但为了好玩，我们故意把扫描码的低 7 位直接送过去了。因此本段代码只供演示，实际应用时还需修改和扩充。

```
// get ascii of make code
reg [6:0] cram_di; // ascii
reg cram_w;
reg kbd_rdn;
always @ (posedge clk) begin
    if (ready) begin
        kbd_rdn <= 0;
        if ((state == 0) && (!is_shift)) begin
            cram_di <= s2a(data); // ascii
            cram_w  <= 1;         // write to char RAM
        end
    end else begin
        cram_w <= 0;
        kbd_rdn <= 1;
    end
end
// scan code to ascii code
function [6:0] s2a;
    input [7:0] s;
    case (s)
        8'h4b: s2a = shift_pressed ? 7'h4c : 7'h6c; // L : l
        8'h43: s2a = shift_pressed ? 7'h49 : 7'h69; // I : i
        8'h35: s2a = shift_pressed ? 7'h59 : 7'h79; // Y : y
        8'h1c: s2a = shift_pressed ? 7'h41 : 7'h61; // A : a
        8'h3a: s2a = shift_pressed ? 7'h4d : 7'h6d; // M : m
        8'h31: s2a = shift_pressed ? 7'h4e : 7'h6e; // N : n
        8'h16: s2a = shift_pressed ? 7'h21 : 7'h31; // ! : 1
        8'h29: s2a = 7'h20; // <space>
        default: s2a = s[6:0]; // just for fun
    endcase
endfunction
```

以下代码产生 kbd_row 和 kbd_col，即当键盘字符要写入字符显示缓冲区时的行地址和列地址。当字符的列位置超过 80 时，显示到下一行的开始处。但如果字符的行位置超过 60 时，本代码没做任何处理 (应该使屏幕图像上滚，即移动字符显示缓冲区的内容)。

```
// character row and column for writing to char RAM
reg [5:0] kbd_row = 0;
reg [6:0] kbd_col = 0;
always @ (negedge cram_w) begin
```

```
        if (kbd_col == 7'd79) begin
            kbd_row <= kbd_row + 6'b1;
            kbd_col <= 0;
        end else kbd_col <= kbd_col + 7'b1;
    end
endmodule
```

以下是 char_ram 模块的具体代码，就是定义一个 8K×7 的存储器。

```
module char_ram (dataout, datain, addr, we, memclk);
    input  [6:0]  datain; // ascii code
    input  [12:0] addr;   // 80 x 60 = 4800 --> 8192 chars
    input         we, memclk;
    output [6:0]  dataout;
    lpm_ram_dq ram (.data(datain), .address(addr), .we(we),
                    .inclock(memclk), .q(dataout));
    defparam      ram.lpm_width   = 7;
    defparam      ram.lpm_widthad = 13;
    defparam      ram.lpm_indata  = "registered";
    defparam      ram.lpm_outdata = "unregistered";
    defparam      ram.lpm_address_control = "registered";
endmodule
```

以下是 font_table 模块的具体代码，就是定义一个 8K×1 的 ROM，其内容是 ASCII 字符的字模。初始化文件是 ascii_font.mif。

```
module font_table (address, clock, q);
    input [12:0]  address; // [12:6] ascii code; [5:3] row [2:0] col;
    input clock;
    output q;
    lpm_rom  lpm_rom_component (.address(address),
                                .inclock(clock), .q(q));
    defparam lpm_rom_component.lpm_width    = 1,
             lpm_rom_component.lpm_widthad  = 13,
             lpm_rom_component.lpm_numwords = "unused",
             lpm_rom_component.lpm_file     = "ascii_font.mif",
             lpm_rom_component.lpm_indata   = "unused",
             lpm_rom_component.lpm_outdata  = "unregistered",
             lpm_rom_component.lpm_address_control = "registered";
endmodule
```

以上代码中调用的 vga 模块和 ps2_keyboard 模块的具体的 Verilog HDL 代码已经给过了。font_table 模块中的初始化文件 (ascii_font.mif) 可根据 font8.c 产生。以下的 C 程序的例子是产生方法之一。

```
extern unsigned char Font[][8];
```

```
main() {
    int chars_per_line = 80; // 80 x 60 characters for VGA
    int char_no;
    unsigned char char_row_bitmap;
    int addr;
    int i, j;
    printf ("DEPTH = 8192;\n");
    printf ("WIDTH = 1;\n");
    printf ("ADDRESS_RADIX = HEX;\n");
    printf ("DATA_RADIX = HEX;\n");
    printf ("CONTENT BEGIN\n");
    addr = 0;
    for (char_no = 0; char_no < 128; char_no++) {
        for (i = 0; i < 8; i++) { // 8 rows per char
            if (char_no >= 0x20) { // <space>
                char_row_bitmap = Font[char_no - 0x20][i];
            } else {
                char_row_bitmap = 0;
            }
            for (j = 7; j >= 0; j--) { // 8 pixels per row
                printf ("%04x : ", addr);
                addr++;
                if (((char_row_bitmap >> j) & 1) == 1) {
                    printf ("1;\n");
                } else {
                    printf ("0;\n");
                }
            }
        }
    }
    printf ("END ;\n");
}
```

15.6 I/O 总线

CPU 与 I/O 设备之间的通信往往通过总线 (Bus) 进行。本节描述串行总线 I2C 和并行总线 PCI，并给出 Verilog HDL 的例子以演示总线接口的电路实现。

15.6.1 I2C 串行总线

I2C 或 I^2C 总线 (Inter Integrated Circuit Bus) 是用于集成电路器件 (I/O 设备或控制器) 之间连接的串行总线。它只有两个信号：SCL (时钟) 和 SDA (数据)。这两个信号都是双向的。当没有器件驱动时，由于提拉电阻的作用，信号为高电平。I2C 允许

有多个主器件 (Master)。在以下的讨论中，我们假定只有一个主器件，其他的全部是从器件 (Slave)。有关 I2C 的完整描述，请阅读文献 [30]。

I2C 总线的传输速率是从 100Kb/s (标准) – 400Kb/s (中速) – 1Mb/s (快速) – 到 3.4Mb/s (高速)。数据信号和时钟信号之间的时序关系如图 15.27 所示。在 SLC 的低电平期间，SDA 可以变化；在 SCL 高电平期间，SDA 必须是稳定的。

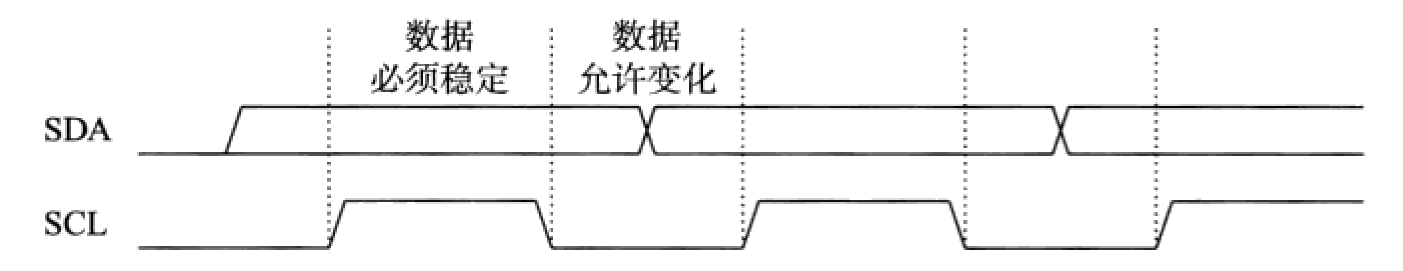


图 15.27 I2C 总线数据信号和时钟信号的时序关系

一次数据的传送由主器件主导。主器件和从器件既可以是发送器也可以是接收器。比如主器件往从器件写数据时，主器件是发送器，从器件是接收器；而主器件接收从器件发来的数据时，主器件是接收器，从器件是发送器。

I2C 的数据传输以字节为单位进行，每次可以传输多个字节。传输开始时，由主器件送出起始位 start，结束时仍是由主器件送出停止信号 stop。起始位是在 SCL 高电平时由 SDA 送出一个下降沿；停止信号是在 SCL 高电平时由 SDA 送出一个上升沿，如图 15.28 所示。

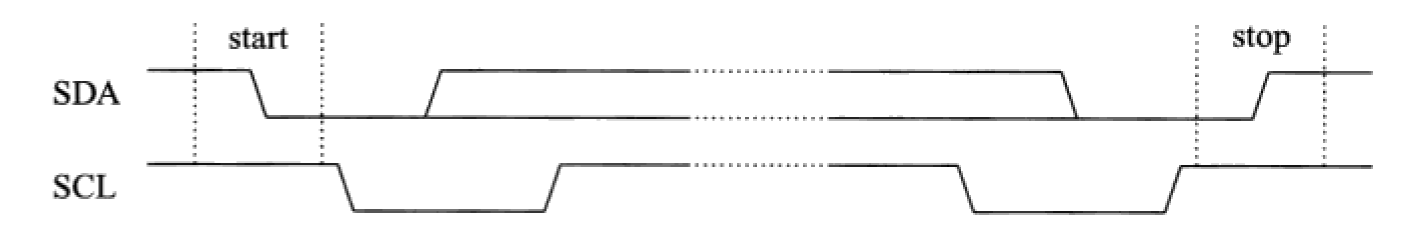


图 15.28 I2C 总线的 start 和 stop 时序的约定

当一个字节由发送器送出后，发送器释放 SDA，由接收器送出一个低电平的回答信号 ack (Acknowledge)，表示已经收到一个字节，如图 15.29 所示。

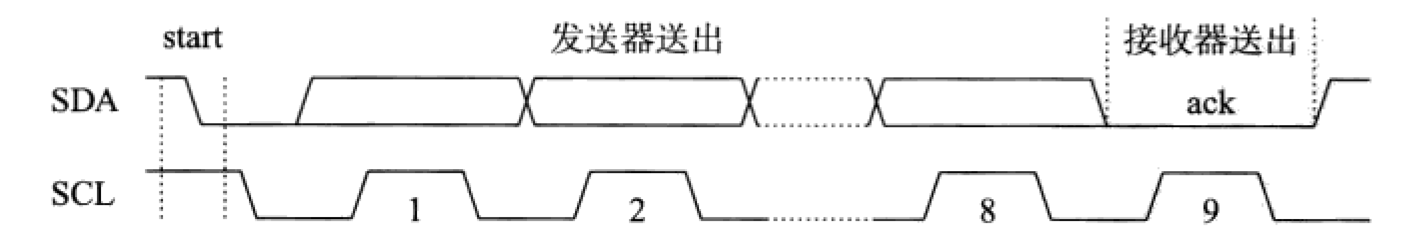


图 15.29 I2C 总线的 ack 时序的约定

所有的时钟脉冲，包括用于回答信号的第 9 个脉冲，都由主器件送出。I2C 总线上可以有多个从器件，每个从器件都有至少一个自己的地址。主器件可以首先送出地址到 I2C 总线上。只有被寻址到的从器件才能送出 ack。主器件可以通过检查是否接收到 ack 来判断从器件是否存在。

如图 15.30 所示，一次送出的从器件的地址有 7 位，故如果地址只送一次，从器件的数量不能超过 128 个。为了消除这个限制，I2C 规定了某些特定的地址可以扩充至 10 位，分两次送出。本书不描述它，只假定地址是 7 位。一个字节有 8 位，除了 7 位地址，剩下的一位用于读写控制 (r/$\overline{\text{w}}$)。当 r/$\overline{\text{w}}$ 为 1 时，表示对从器件进行读操作；为 0 时，表示写操作。注意，只有跟在起始位 start 后面的字节才做如此解释，其他的作为 8 位数据看待。

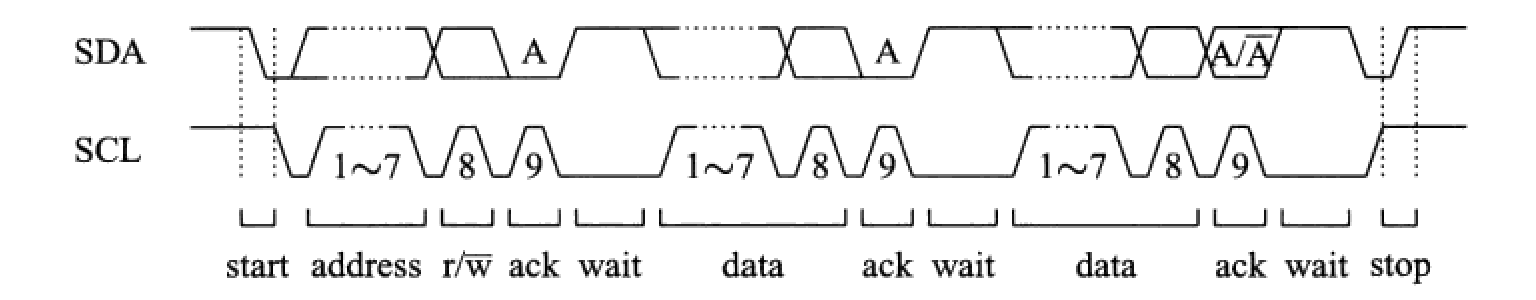


图 15.30 I2C 总线的地址、读写和数据时序的约定

一个字节的发送从最高位开始。当主器件为回答信号送出第 9 个时钟正脉冲后，释放 SCL。这时从器件如果要做内部处理而没有能力继续接收或发送，就强迫 SCL 变低电平，告诉主器件先等一等 (图 15.30 中的 wait)。当从器件释放 SCL 后，字节传输可以继续进行。

并不是发送器总是能收到一个低电平的回答信号，有时会收到一个高电平。我们称这个高电平为 nack (Not Acknowledge)，即不回答。原因有以下 4 种：(1) 接收器根本就不存在；(2) 接收器正忙着呢；(3) 收到的数据或命令看不懂；(4) 主器件 (接收器) 命令从器件停下来别送了。图 15.30 中的 $\overline{\text{A}}$ 就是 nack。

因为 I2C 上连接有多个从器件，主器件决定要和哪个从器件开始通信时，必须首先送出那个从器件的地址，并告诉它是读还是写。图 15.31 是主器件往从器件写数据的例子 (主器件是发送器)。图中的深色部分由主器件送出。S 代表起始位，Slave Address 是从器件的 7 位地址，r/$\overline{\text{w}}$ 为 0，表示写操作，然后连续送两个字节的数据，最后送出停止位 (图中用 P 表示)。浅色部分由从器件送出，每收到一个字节，都送出一个回答。最后一个也可能不回答。

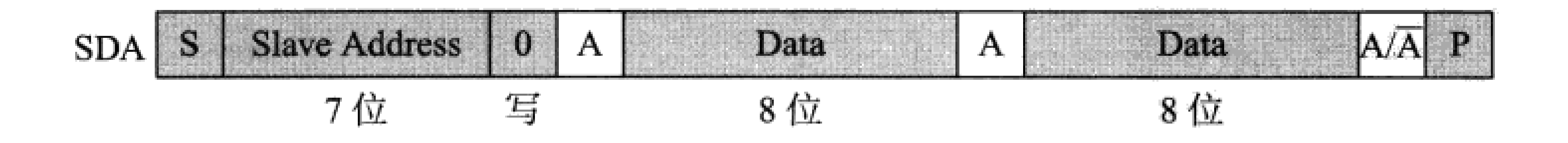


图 15.31 I2C 总线的写时序

图 15.32 是主器件从从器件读数据的例子 (主器件是接收器)。首先由主器件送出起始位、从器件地址以及 r/$\overline{\text{w}}$ 位 (为 1，表示是读操作)。从器件给出回答，送出第一个字节的数据。主器件收到这个字节后，给一个回答。从器件再送出第二个字节的数据。然后主器件说，够了，别送了 ($\overline{\text{A}}$)，并告诉从器件：本次读操作结束 (P)。

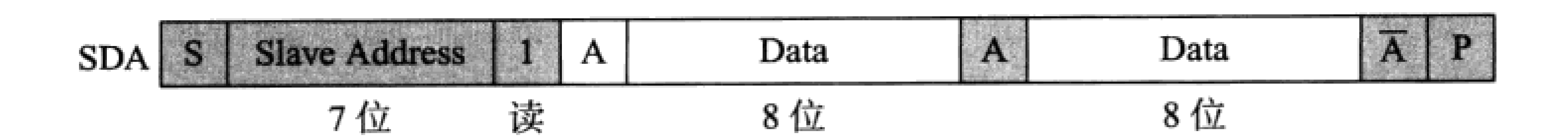


图 15.32 I2C 总线的读时序

以上的两个图分别是写操作和读操作的时序。在主器件和从器件的一次通信过程期间，可以完成读和写的混合操作，如图 15.33 所示。主器件先往从器件写数据。这个数据的意义可能是从器件的内部寄存器号，也可能是从器件内部的存储器 (比如 EPROM) 地址。然后再送出起始位，开始读操作。注意，在这个起始位之前，主器件并没有送出停止位。第二个起始位称为重复的起始位 Sr (Repeated Start)。

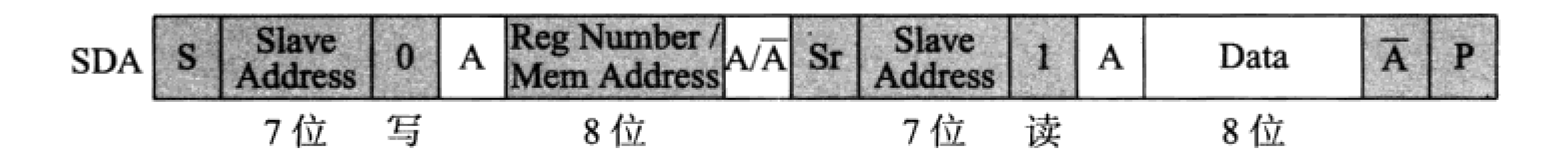


图 15.33 I2C 总线的先写后读时序

I2C 总线的基本的协议就简单地介绍到这里，以下演示如何用 Verilog HDL 实现 I2C 总线的接口电路。注意，以下的代码并非针对某种特定的 I2C I/O 设备，而是尽可能给出比较通用的代码。

假设系统时钟 clk 是 50MHz，I2C 总线的传输速率是 400kb/s。我们把 clk 25 分频，生成 2MHz 的 I2C 时钟 i2c_clk，这样每发送一位需要 5 个 i2c_clk (2M/5 = 400k)。图 15.34 示出的是使用 5 个 i2c_clk 发送 start、1、0 和 stop 的波形。scl 有两个周期的高电平和 3 个周期的低电平，接收数据时在 i2c_clk 的第 2、3 周期之间的上升沿处采样 0 或 1 数据，采到小圆圈处的数据。

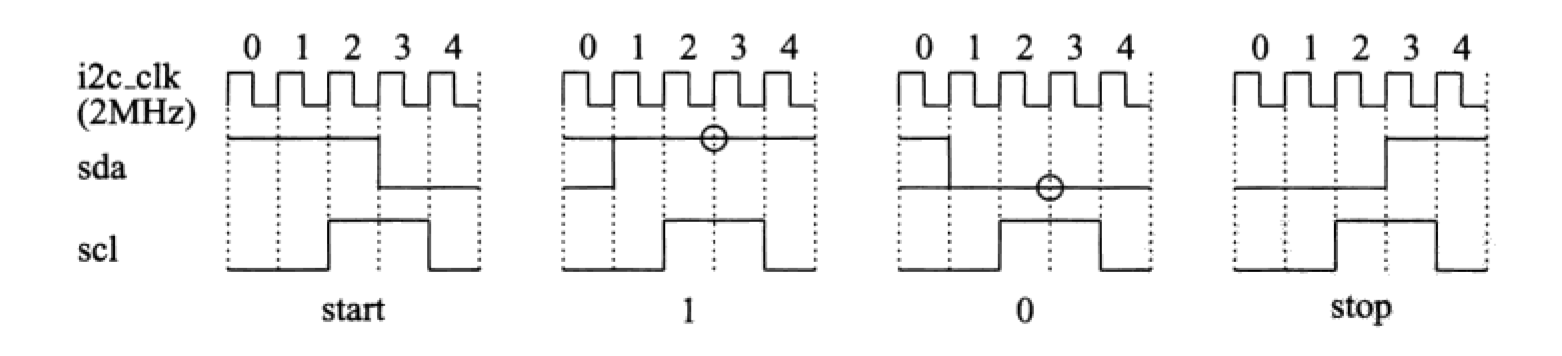
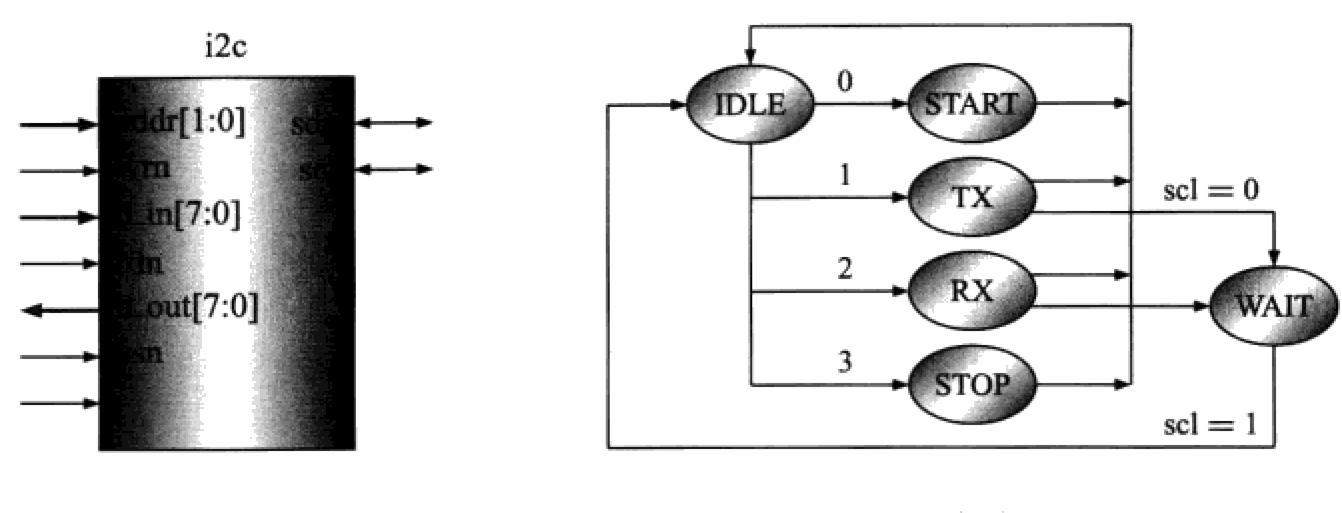


图 15.34 I2C 总线信号和 i2c_clk 时钟信号之关系

为发送和接收回答信息而送出的 scl 稍有不同 (图中没有画出)，它的高电平有 3 个 i2c_clk 周期 (最后一个周期也为高，以检查是否要等待)。

图 15.35 是 I2C 总线接口控制器的信号和状态转移图。图中的 clk 是系统时钟；csn 是低电平有效的片选信号；addr[1:0] 是地址；wrn 是低电平有效的写信号；d_in[7:0] 是输入数据；rdn 是低电平有效的读信号；d_out[7:0] 是输出数据。



(a) I2C 总线接口控制器信号　　(b) 状态转移图

图 15.35　I2C 总线接口控制器信号和状态转移图

状态转移图一共有 6 个状态。IDLE 状态等待 wrn 和 addr[1:0]，以便转至下一状态。如果下一状态要发送，则需把要发送的数据 (d_in[7:0]) 锁存下来。如果是接收，则要把回答信号 (ack/nack) 锁存下来 (d_in[0])，详细的动作见表 15.3；START 状态送出起始位；TX 状态发送 8 位数据并接收回答，如果 scl 信号被从器件强制拉低，则进入等待状态；RX 状态接收 8 位数据并送出回答，如果 scl 信号被从器件强制拉低，则进入等待状态；STOP 状态送出停止位；WAIT 状态简单地等待 scl 被从器件释放。从 IDLE 转向哪个状态由驱动程序决定，方法是送不同的 addr 给 i2c 控制器 (wrn = 0)，见表 15.3 中的定义。所以，这是一个通用的状态转移图。针对不同的 I2C I/O 器件，我们可以书写不同的驱动程序，使用同一个 I2C 总线控制器，实现主器件和各种从器件之间的通信。

表 15.3　总线主器件控制器地址 addr[1:0] 的意义

addr[1:0]	rdn = 0	wrn = 0
0	读数据	转至 START 状态
1	读状态	转至 TX 状态，同时锁存 d_in[7:0]
2		转至 RX 状态，同时锁存 d_in[0] (ack/nack)
3		转至 STOP 状态

以下是 I2C 总线主器件控制器的 Verilog HDL 代码。我们把模块内部的一些信号拉到了外面，供测试用。实际的信号见图 15.35。我们对整个代码分段加以说明。

```
`define IDLE  0
`define START 1
`define TX    2
`define RX    3
`define STOP  4
`define WAIT  5
module i2c (clk,csn,addr,wrn,d_in,rdn,d_out,sda,scl,i2c_clk,
            pulse_count,bit_count,curr_state,next_state);
```

```
input  clk;          // system clock, 50MHz
input  csn;          // chip select, active low
input  [1:0] addr;   // address
input  wrn;          // write, active low
input  [7:0] d_in;   // data in, data to be sent
input  rdn;          // read, active low
output [7:0] d_out;  // data out, received data or status
inout  sda;          // I2C SDA
inout  scl;          // I2C SCL
output i2c_clk;             // for test
output [2:0] pulse_count;   // for test
output [3:0] bit_count;     // for test
output [2:0] curr_state;    // for test
output [2:0] next_state;    // for test
```

以下代码生成 2MHz 的时钟信号 i2c_clk。本应对 50MHz 的 clk 进行 25 分频，但代码中故意只做了 4 分频，只是为了看波形图方便起见。读者可以把代码中的 3 和 1 分别用 24 和 12 (或 11) 替换掉 (在代码中做了标注)。

```
// I2C clock
reg [4:0] clk_count = 0;   // clk / 25 = 2MHz
reg       i2c_clk   = 1;   // 2MHz
always @(posedge clk) begin
    if  (clk_count == 3) clk_count <= 0; // (clk_count == 24)
    else clk_count <= clk_count + 5'd1;
    if  (clk_count <= 1) i2c_clk <= 1;   // (clk_count <= 12)
    else                 i2c_clk <= 0;
end
```

以下代码生成两个计数器。一个是 pulse_count，对 i2c_clk 计数，范围是 0 ~ 4，其用途见图 15.34；另一个是 bit_count，对数据位计数，范围是 0 ~ 8。该计数器只为发送或接收状态服务。

```
// pulse_count and bit_count
reg [2:0] pulse_count = 0; // counting i2c_clk cycles
reg [3:0] bit_count   = 0; // counting number of bits
always @(posedge i2c_clk) begin
    if (curr_state == `IDLE) begin
        pulse_count <= 0;
        bit_count   <= 0;
    end else begin
        if (pulse_count == 4) pulse_count <= 0;
        else if ((curr_state != `WAIT) || scl)
            pulse_count <= pulse_count + 3'd1;
        if (((curr_state == `TX) || (curr_state == `RX)) &&
             (pulse_count == 4)) begin
            if  (bit_count == 8) bit_count <= 0;
```

```
            else bit_count <= bit_count + 4'd1;
        end
    end
end
```

以下代码实现状态转移，请参考图 15.35 及表 15.3。注意，为了节省版面，我们把 case 语句内部的 end 放在一个语句的右边，而不是占用一行。

```
// next state and data input
reg [7:0] in_buf;              // data to be sent
reg       tx_ack;              // ack sent be master
reg [2:0] curr_state = `IDLE; // current state
reg [2:0] next_state = `IDLE; // next state
always @(posedge clk) begin
    case (curr_state)
        `IDLE:  begin
            if ((!csn) && (!wrn)) begin
                case (addr)
                    2'd0: begin next_state <= `START; end
                    2'd1: begin next_state <= `TX;
                                in_buf <= d_in;       end
                    2'd2: begin next_state <= `RX;
                                tx_ack <= d_in[0];    end
                    2'd3: begin next_state <= `STOP;  end
                    default:    next_state <= `IDLE;
                endcase
            end end
        `START: if (pulse_count == 4) next_state <= `IDLE;
        `TX:    if (bit_count == 8)
                   case (pulse_count)
                      3'd2: if (scl == 0) next_state <= `WAIT;
                      3'd3: if (scl == 0) next_state <= `WAIT;
                      3'd4: if (scl == 0) next_state <= `WAIT;
                            else          next_state <= `IDLE;
                      default: ;
                   endcase
        `RX:    if (bit_count == 8)
                   case (pulse_count)
                      3'd2: if (scl == 0) next_state <= `WAIT;
                      3'd3: if (scl == 0) next_state <= `WAIT;
                      3'd4: if (scl == 0) next_state <= `WAIT;
                            else          next_state <= `IDLE;
                      default: ;
                   endcase
        `STOP:  if (pulse_count == 4) next_state <= `IDLE;
        `WAIT:  if (scl != 0)         next_state <= `IDLE;
```

```
        endcase
    end
    always @(posedge i2c_clk) begin
        curr_state <= next_state;
    end
```

以下代码实现数据 (也包括起始位和停止位) 的发送和接收。由于 I2C 总线信号是双向的，我们必须使用三态门。两个信号 scl 和 sda 的三态门控制信号分别是 enable_scl 和 enable_sda，见开始处的两个 assign 语句。如果想送出一位 1，则令三态门控制信号为 1，送出高阻 (相当于 1)；如果想送 0，则令三态门控制信号为 0，送出 0。重要部分是使用 pulse_count 来控制一位数据的发送时序和接收采样点，使用 bit_count 来控制一个字节及回答位的发送与接收。发送字节时要接收回答信息 (ack/nack)，接收字节时要发送回答信息。注意为了接收回答信息而送出的 scl (由 enable_scl 控制) 与其他情况下的 scl 稍有不同。

```
    // transfer data via I2C bus
    assign scl = enable_scl ? 1'bz : 1'b0;
    assign sda = enable_sda ? 1'bz : 1'b0;
    reg [7:0] out_buf;          // data received
    reg       rx_ack;           // ack received
    reg       txd;              // bit to be sent
    reg       enable_scl = 0;  // tri-state control
    reg       enable_sda = 0;  // tri-state control
    always @(posedge i2c_clk) begin
        case (curr_state)
            `IDLE:  begin enable_scl <= 0; enable_sda <= 0; end
            `START: begin
                case (pulse_count)
                    3'd0: begin enable_scl <= 0;
                                enable_sda <= 1; end
                    3'd1: begin enable_scl <= 1;
                                enable_sda <= 1; end
                    3'd2: begin enable_scl <= 1;
                                enable_sda <= 0; end
                    3'd3: begin enable_scl <= 0;
                                enable_sda <= 0; end
                    3'd4: begin enable_scl <= 0;
                                enable_sda <= 0; end
                endcase end
            `TX: begin
                if (bit_count == 8) begin // receive ack/nack
                    case (pulse_count)
                        3'd0: begin enable_scl <= 0;
                                    enable_sda <= 1; end
                        3'd1: begin enable_scl <= 1;
```

```
                        enable_sda <= 1; end
            3'd2: begin enable_scl <= 1;
                        enable_sda <= 1;
                        rx_ack <= sda;   end
            3'd3: begin enable_scl <= 1;
                        enable_sda <= 1; end
            3'd4: begin enable_scl <= 0;
                        enable_sda <= 1; end
        endcase
    end else begin // send data bit
        case (pulse_count)
            3'd0: begin
                enable_scl <= 0;
                enable_sda <= in_buf[7-bit_count]; end
            3'd1: begin
                enable_scl <= 1;
                enable_sda <= in_buf[7-bit_count]; end
            3'd2: begin
                enable_scl <= 1;
                enable_sda <= in_buf[7-bit_count]; end
            3'd3: begin
                enable_scl <= 0;
                enable_sda <= in_buf[7-bit_count]; end
            3'd4: begin
                enable_scl <= 0;
                enable_sda <= in_buf[7-bit_count]; end
        endcase
    end end
`RX: begin
    if (bit_count == 8) begin // send ack/nack
        case (pulse_count)
            3'd0: begin enable_scl <= 0;
                        enable_sda <= tx_ack; end
            3'd1: begin enable_scl <= 1;
                        enable_sda <= tx_ack; end
            3'd2: begin enable_scl <= 1;
                        enable_sda <= tx_ack; end
            3'd3: begin enable_scl <= 1;
                        enable_sda <= tx_ack; end
            3'd4: begin enable_scl <= 0;
                        enable_sda <= tx_ack; end
        endcase
    end else begin // receive data bit
        case (pulse_count)
            3'd0: begin enable_scl <= 0;
```

```
                                enable_sda <= 1; end
                     3'd1: begin enable_scl <= 1;
                                enable_sda <= 1; end
                     3'd2: begin enable_scl <= 1;
                                enable_sda <= 1;
                                out_buf[7-bit_count] <= sda;
                                end
                     3'd3: begin enable_scl <= 0;
                                enable_sda <= 1; end
                     3'd4: begin enable_scl <= 0;
                                enable_sda <= 1; end
                 endcase
             end end
         `STOP: begin
             case (pulse_count)
                 3'd0: begin enable_scl <= 0;
                             enable_sda <= 0; end
                 3'd1: begin enable_scl <= 1;
                             enable_sda <= 0; end
                 3'd2: begin enable_scl <= 1;
                             enable_sda <= 1; end
                 3'd3: begin enable_scl <= 0;
                             enable_sda <= 1; end
                 3'd4: begin enable_scl <= 0;
                             enable_sda <= 1; end
             endcase end
         `WAIT:        begin enable_scl <= 1;
                             enable_sda <= 1; end
     endcase
 end
```

以下代码为 CPU 提供接收到的数据或者总线控制器本身的状态信息。

```
    // read from host
    reg [7:0] d_out;
    always @(posedge clk) begin
        if ((!csn) && (!rdn)) begin
            case (addr)
                2'd0: d_out <= out_buf;
                2'd1: d_out <= {enable_scl, enable_sda, scl, sda,
                                curr_state, rx_ack};
                default: d_out <= 8'dz;
            endcase
        end
    end
endmodule
```

代码到此结束，还算简洁吧。但简洁不是它的优点，优点是灵活。灵活的意思是通用：你只要书写驱动程序，使用以上电路基本上可以和任何 I2C I/O 器件通信。

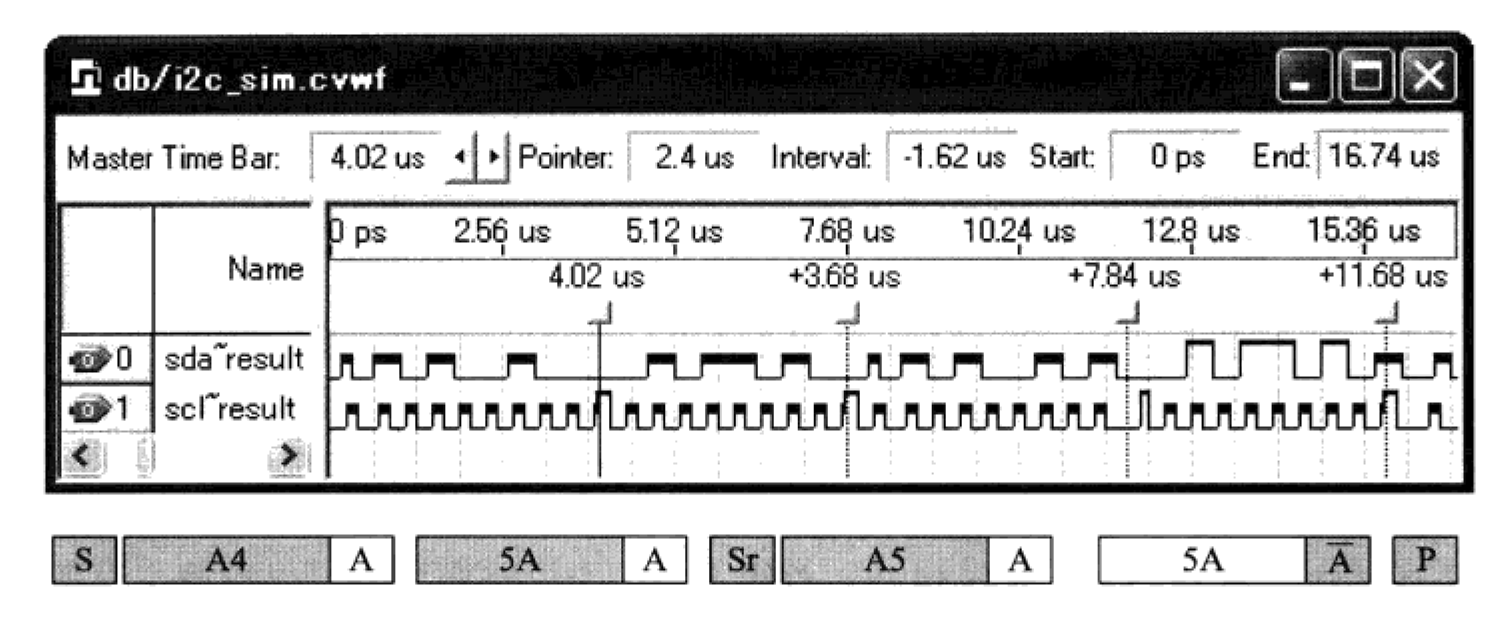


图 15.36 I2C 仿真波形 (全时段) 及数据

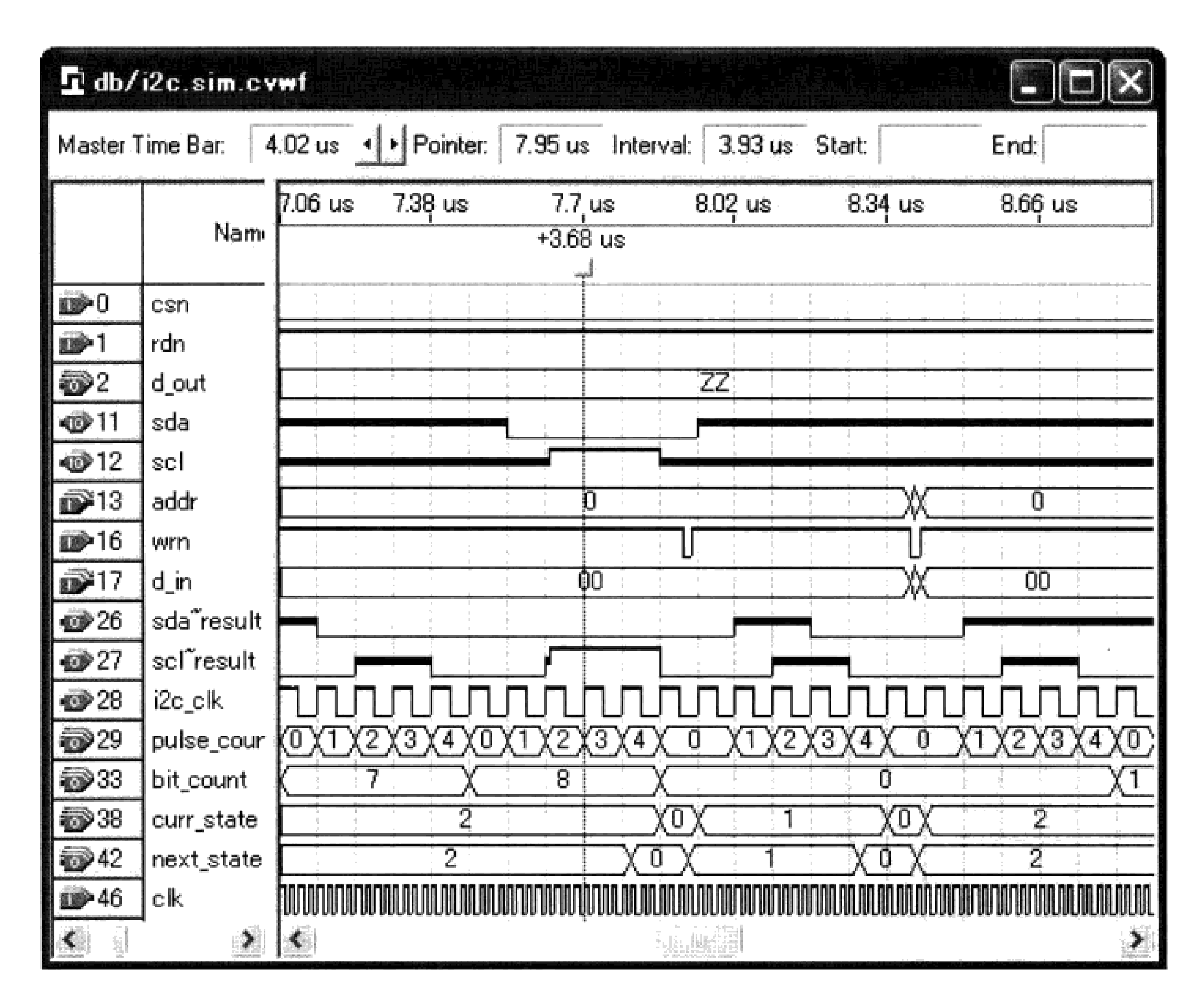


图 15.37 I2C 仿真波形

图 15.36 是仿真波形，给出类似于图 15.33 的时序，即先写后读。仿真波形的数据在波形图的下方给出，前三个字节由 I2C 控制器送出，最后一个字节从 I2C I/O 设

备读出。字节及 S (Sr) 之间的空隙表示 IDLE 状态。第三个字节送出后进入等待状态，其余都没等待。注意双向信号 scl 和 sda 的输出分别用 scl~result 和 sda~result 表示。不高也不算低的波形表示 Z (高阻) 状态，由于外部接有提拉电阻，它实际上可以被认为是 1。图 15.37 是发送 Sr 前后的比较详细的波形图。请读者对照源代码仔细研究一下波形，说不定会发现一些错误。

15.6.2 PCI 并行总线

PCI (Peripheral Component Interconnect) 是一种并行的高性能总线，用于连接各种扩展电路板和 CPU/存储器子系统等，目前被 PC、工作站和伺服器所广泛使用。PCI 总线是一种同步的分时复用的双向总线 (地址和数据共用相同的信号线)。

按 PCI 的约定，数据传输的发起方 (Initiator) 是主设备 (Master)，被动的一方 (Target) 是从设备 (Slave)。连接到 PCI 总线上的一个电路模块既可以扮演主设备的角色，也可以扮演从设备的角色。即，PCI 允许有多个总线控制器。当然，一个电路模块只扮演从设备的角色也未尝不可。图 15.38 示出了 PCI 的总线信号。

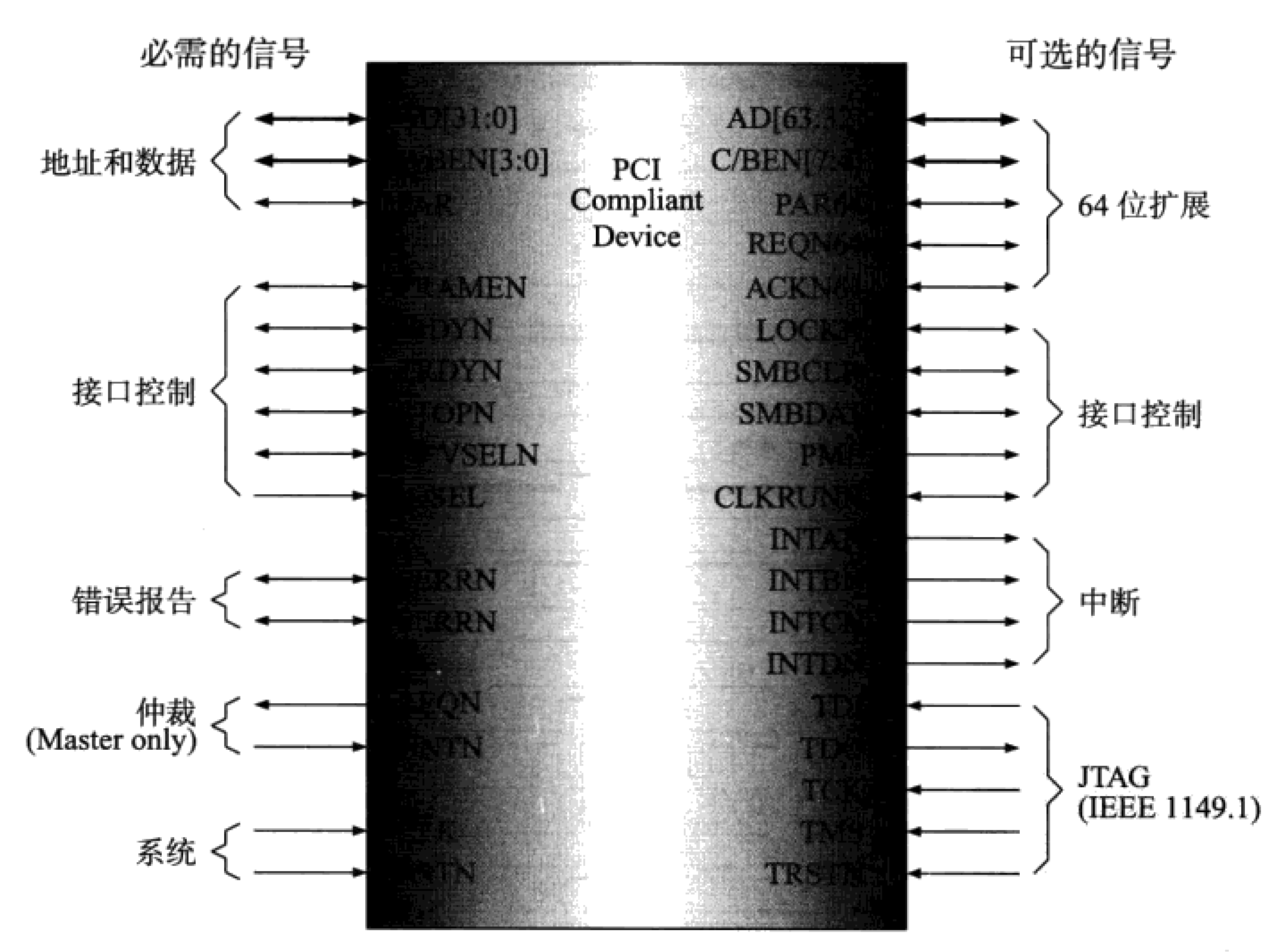


图 15.38 PCI 总线信号

图中左侧信号是必须要有的，右侧是供选择用的。主设备必须具备的信号有 49 个。如果只扮演从设备的角色，需 47 个信号 (没有 REQN 和 GNTN)。以下简要描述必需信号的意义，详细的描述请参阅文献 [28]。注意信号名称以 “N” 结尾的表示低电平有效 (在 PCI 标准中使用 #)。

1) CLK: 时钟 (Clock) 信号。除了复位和中断信号外，其他的均是以 CLK 为基准的同步信号：CLK 的上升沿对输入信号采样，输出信号在下降沿处送出。
2) RSTN: 低电平有效的复位 (Reset) 信号。用于所有 PCI 设备的内部状态及输出信号的复位。在 CLK 信号稳定之后，RSTN 至少要保持 100ms 有效。
3) AD[31:0]: 分时复用的地址和数据 (Address and Data) 总线。地址和数据共用总线是为了减少引脚的数量。一次总线传输以一个地址周期 (Address Phase) 开始，跟着一个或多个数据周期 (Data Phase)。多个数据的地址自动递增。
4) C/BEN[3:0]: 总线命令和字节使能 (Bus Command and Byte Enables)。在地址周期，C/BEN[3:0] 指出总线传输的类型 (存储器或 I/O 读写命令等)，见表 15.4；在数据周期，C/BEN[3:0] 对应 AD[31:0] 上四个字节的使能 (低电平有效)，BEN[3] 对应最高字节，BEN[0] 对应最低字节。

表 15.4 C/BEN[3:0] 之命令类型

C/BEN[3:0]	命令类型	C/BEN[3:0]	命令类型
0000	中断响应	1000	保留
0001	特殊周期	1001	保留
0010	I/O 读	1010	配置读
0011	I/O 写	1011	配置写
0100	保留	1100	读多个 Cache 块
0101	保留	1101	64 位地址周期
0110	存储器读	1110	读一个 Cache 块
0111	存储器写	1111	写一个 Cache 块

5) PAR: 对 AD[31:0] 和 C/BEN[3:0] 的偶校验位 (Parity)，即 AD[31:0]、C/BEN[3:0] 和 PAR 合在一起为 1 的位数是偶数。PAR 信号的时序与 AD[31:0] 相同，但比 AD[31:0] 晚一个时钟周期，以便有充裕的时间来计算校验位。
6) IRDYN: 低电平有效的主设备准备好 (Initiator Ready)。主设备往从设备写数据时，主设备发出 IRDYN 表示它已经把数据放在 AD[31:0] 上了。读数据时，主设备发出 IRDYN 表示它已经准备好了接收数据。IRDYN 要保持有效，直到 TRDYN 有效或从设备发出停止信号 STOPN 为止。
7) TRDYN: 低电平有效的从设备准备好 (Target Ready)。主设备往从设备写数据时，从设备发出 TRDYN 表示它已经准备好了接收数据。读数据时，从设备发出 TRDYN 表示它已经把数据放在 AD[31:0] 上了。只有当 IRDYN 和 TRDYN 都有效时才传输数据，否则等待，以实现不同速度的设备之间的数据传输。
8) FRAMEN: 主设备发出低电平有效的 FRAMEN 表示数据传输开始或正在进行。撤销 FRAMEN 表示数据传输已是最后一个数据周期或已经结束。
9) DEVSELN: 从设备发出低电平有效的 DEVSELN 表示自己已被选中。主设备可以使用这个输入信号来判断要访问的从设备是否出现在 PCI 总线上。

10) STOPN: 低电平有效的 STOPN 要求主设备停止当前的数据传输。如果从设备需要很长的时间来响应主设备发出的数据传输请求，它会发出 STOPN 信号来暂时“挂起”当前的数据传输，以让总线来完成其他的数据传输。
11) IDSEL: IDSEL (Initialization Device Select) 是一个专门为初始化 PCI 设备而准备的高电平有效的片选信号。在地址周期，当 IDSEL 为高电平并且 AD[1:0] 为 00 时，C/BEN[3:0] 提供初始化命令，AD[10:8] 提供功能码，AD[7:2] 选择 PCI 设备的内部寄存器。
12) REQN: 希望使用 PCI 总线的请求 (Request) 信号，低电平有效。注意只有主设备才有此信号，从设备没有。PCI 总线上允许有多个主设备，每个主设备都有一个 REQN 信号。主机系统中有一个总线仲裁器接收每个主设备的请求信号。复位信号 RSTN 有效时，REQN 必须是高阻状态。
13) GNTN: 总线仲裁器发出的对 REQN 的响应 (Grant) 信号，低电平有效。每个主设备都有一个 GNTN 信号。当 GNTN 持续一个时钟周期有效时，主设备可以在下一个周期发出 FRAMEN，开始使用 PCI 总线传输数据。与 REQN 一样，从设备没有此信号。复位信号 RSTN 有效时，GNTN 必须被忽略。
14) PERRN: 除了“特殊周期”(Special Cycle)，所有其他的传输周期中如果偶校验出错了，PERRN (Parity Error) 送出有效信号。因为 PAR 比 AD[31:0] 晚一个周期，所以 PERRN 要比 AD[31:0] 晚两个周期。这是一个三态信号，低电平有效，无效时浮空。由于该信号外接提拉电阻，从有效到浮空需要较长的时间，因此要求该信号在进入浮空状态之前必须有至少一个时钟周期的高电平。
15) SERRN: 如果特殊周期中地址或数据的偶校验出错，或者出现其他致命的错误时，SERRN (System Error) 送出有效信号。这是一个漏极开路 (Open Drain) 的信号，与三态信号一样外接提拉电阻。

PCI 定义相邻的两个时钟下降沿之间为一个周期，上升沿处在一个周期的中央。图 15.39 示出了 PCI 读操作的时序，每个时钟周期的动作如下所述。

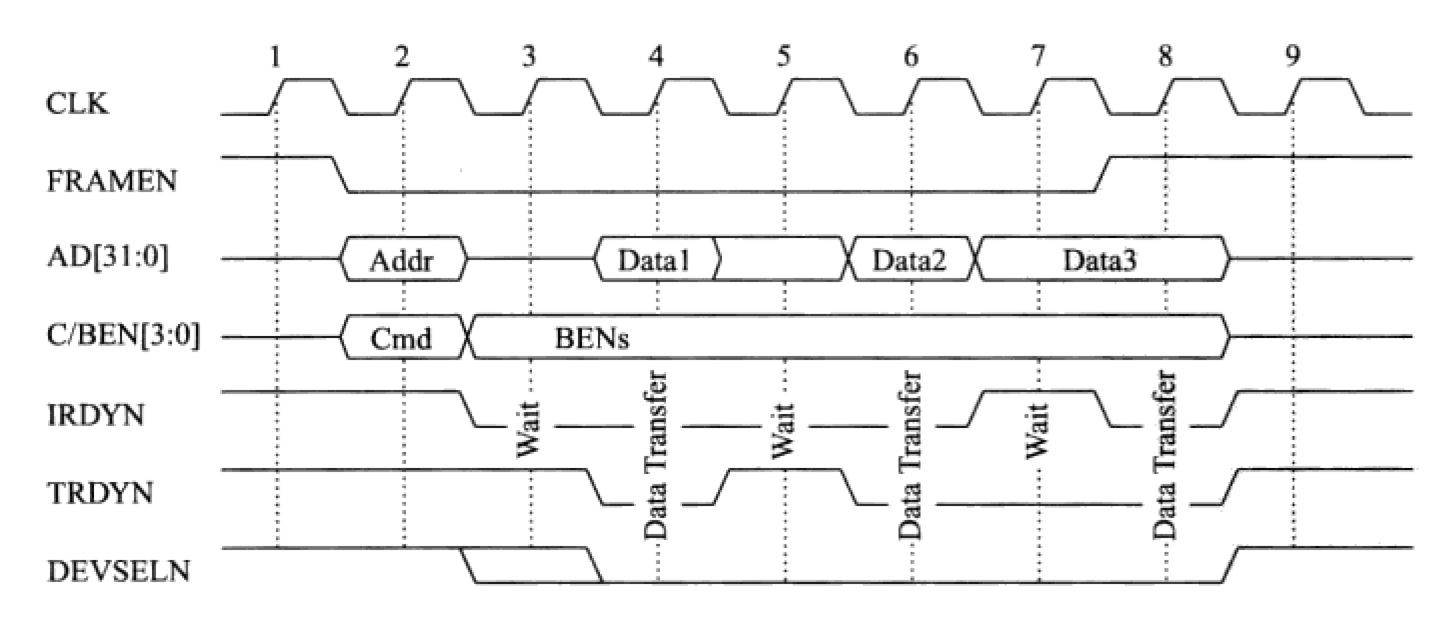


图 15.39 PCI 总线基本读操作时序

- 周期 1 总线空闲。
- 周期 2 主设备发出一个有效地址 AD[31:0] 并由 C/BEN[3:0] 送出读命令。这是一个地址周期。有效的 FRAMEN 也在这个周期发出。
- 周期 3 主设备释放地址线、由 C/BEN[3:0] 送出字节使能并发出有效的 IRDYN，表示它可以接收数据了。作为回答，从设备送出有效的 DEVSELN (或在下一个周期送出)。从设备把 TRDYN 置高表示它还没有提供有效的数据。
- 周期 4 从设备送出数据并拉低 TRDYN，告诉主设备有效的数据已经出现在 AD[31:0] 上了。主设备取走数据。这是第一个数据周期。注意在数据传输期间 IRDYN 和 TRDYN 均为有效的低电平。
- 周期 5 从设备拉高 TRDYN，告诉主设备下一个数据还没准备好。
- 周期 6 是第二个数据周期，IRDYN 和 TRDYN 均有效，主设备取走数据。
- 周期 7 从设备提供第三个数据，TRDYN 也保持有效。但这时主设备没准备好，把 IRDYN 置成了无效。
- 周期 8 主设备重新拉低 IRDYN，取走数据，以结束第三个数据周期。主设备拉高 FRAMEN，指明这是最后一个数据周期。
- 周期 9 所有信号都撤销，回到空闲状态。

图 15.40 示出了 PCI 写操作的时序，每个时钟周期的动作如下所述。

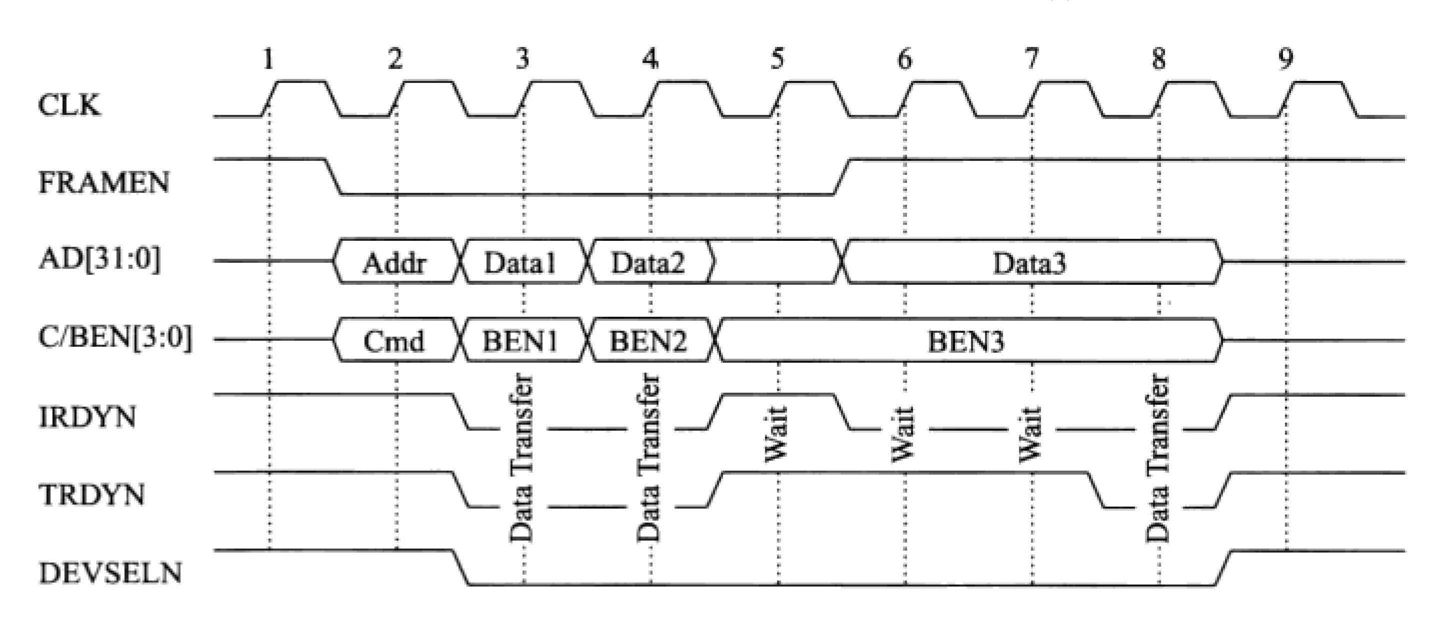


图 15.40 PCI 总线基本写操作时序

- 周期 1 总线空闲。
- 周期 2 主设备发出一个有效地址 AD[31:0] 并由 C/BEN[3:0] 送出写命令。这是一个地址周期。有效的 FRAMEN 也在这个周期发出。
- 周期 3 主设备由 AD[31:0] 送出数据、由 C/BEN[3:0] 送出字节使能并发出有效的 IRDYN，告诉从设备数据有效。从设备送出有效的 DEVSELN 及 TRDYN，取走数据。第一个数据周期结束。
- 周期 4 主设备提供第二个数据及字节使能。IRDYN 和 TRDYN 均为有效的低电平，从设备取走数据，第二个数据周期结束。

- 周期 5 主设备和从设备都没准备好，IRDYN 和 TRDYN 均为高电平。
- 周期 6 主设备提供第三个数据及字节使能，IRDYN 也被拉低，告诉从设备数据有效。主设备撤销 FRAMEN，告诉从设备这是最后一个数据周期。但这时从设备还没准备好，保持 TRDYN 高电平。
- 周期 7 从设备还没准备好，继续保持 TRDYN 高电平。IRDYN 还是有效的低电平，等着 TRDYN。
- 周期 8 从设备终于准备好了，拉低 TRDYN，取走数据。主设备见到有效的 TRDYN，撤销 IRDYN。第三个数据周期结束。
- 周期 9 平安无事，回到空闲状态。

以下我们给出一个简单的 PCI 从设备的设计例子。它实现的功能是通过 PCI 总线来访问存储器，其输入输出信号见图 15.41。注意该例并没有提供 PCI 总线所要求的全部信号。与存储器连接的信号描述如下。(1) mem_addr[31:0] 是 32 位存储器地址，我们假设存储器的高 16 位地址为全 1；(2) mem_data_read[31:0] 是 32 位从存储器读出的数据；(3) mem_data_write[31:0] 是 32 位写入存储器的数据；(4) mem_read_write 是存储器读写信号，1 读 0 写；(5) mem_ready 是存储器准备好信号。

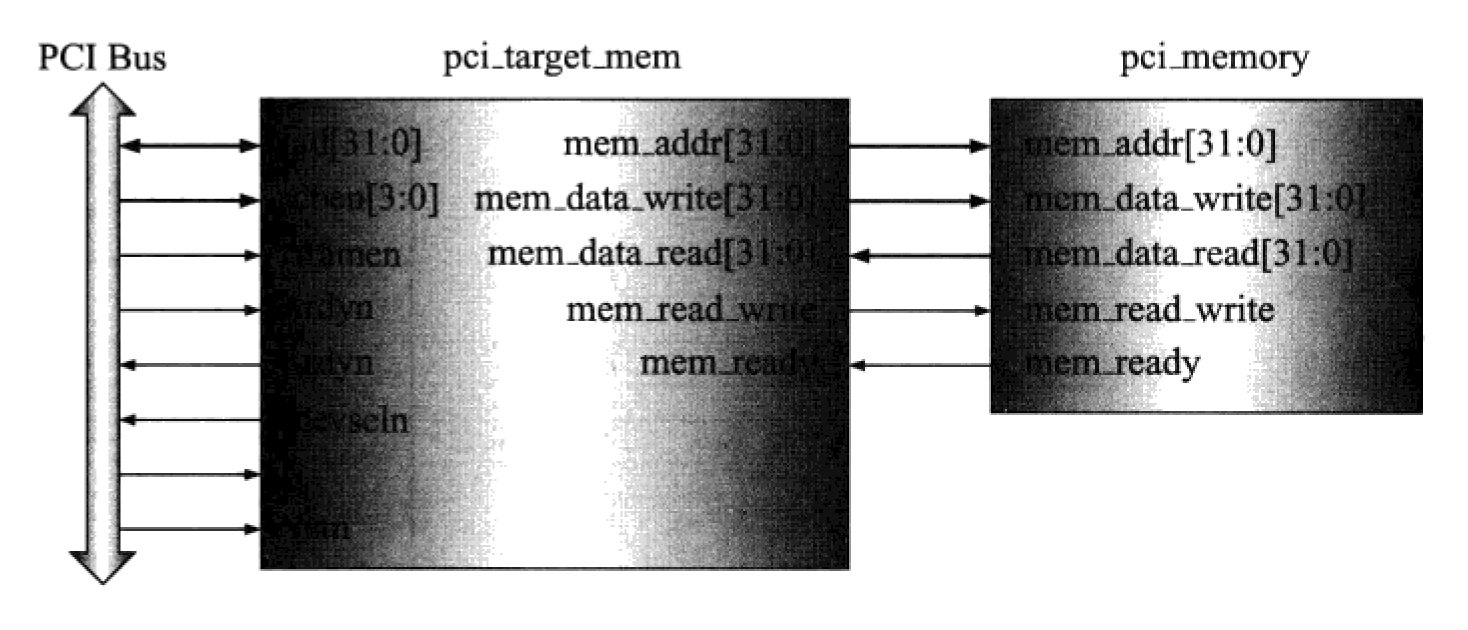


图 15.41 PCI 从设备 (存储器) 输入输出信号

我们将给出图中 pci_target_mem 模块的 Verilog HDL 代码。我们定义了三个状态，它们分别是空闲状态 IDLE、存储器读状态 R_MEM 和存储器写状态 W_MEM。这些状态在 Verilog HDL 的实现代码中用 next_state 表示，状态的转换发生在时钟的上升沿，见图 15.42。例如，当 cben = 7 时，从上升沿开始，next_state = W_MEM。

我们可以把 PCI 总线上的地址和数据先用时钟上升沿锁存下来，再去访问存储器。但这样会引入半个时钟周期的延迟。我们的做法是不锁存，直接用 PCI 总线上的地址和数据访问存储器。因此，我们把 next_state 用时钟的下降沿锁存下来，用 state 表示，并用它来产生存储器的访问控制信号。

我们通过检测 framen 下降沿的办法来判断一次传输是否开始。为此，我们使用了一个内部信号 pre_framen 来记录 framen 在前一个周期的电平。即，当 pre_framen

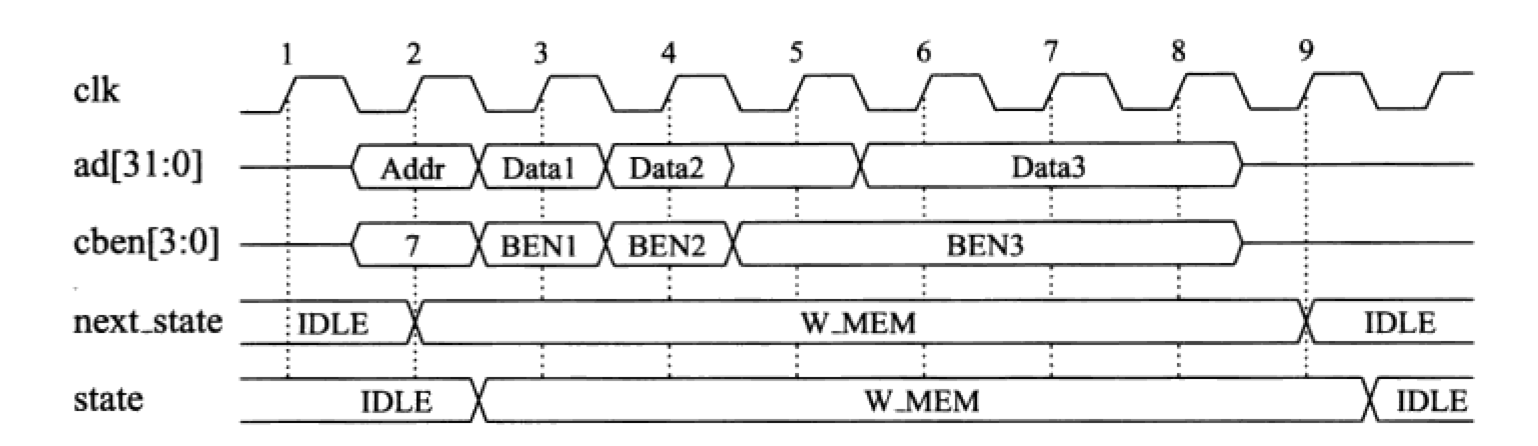


图 15.42 Next_State 和 State (存储器写)

为 1 且 framen 为 0 时，一次传输正式开始。当 irdyn 和 trdyn 均有效时，传输一个数据，存储器地址加 1 (如果使用字节地址则应该加 4)。

注意，如果从写数据 ad 的角度看，状态 state 实际上是晚了一个周期。比如，写第一个数据时的状态 (state = W_MEM) 来自于前一个周期的 cben = 7 (见图 15.42)。如此，在最后一次写数据时，还不能确定写状态在下一状态结束。读数据也是一样。因此我们把输出使能信号 enable (为 1 时驱动 ad 总线) 与上了 $\overline{\text{framen} \cdot \text{irdyn}}$，即，当 framen 和 irdyn 均无效时，令 enable 为 0。若非如此，会和 PCI 主设备驱动 ad 总线发生冲突。以下是实现 PCI 存储器读写的 Verilog HDL 代码。

```
`define IDLE   0
`define R_MEM  1
`define W_MEM  2
module pci_target_mem (clk,rstn,framen,cben,ad,irdyn,trdyn,
                       devseln,mem_read_write,mem_ready,mem_addr,
                       mem_data_write,mem_data_read,state);
    input  clk;           // clock
    input  rstn;          // reset
    input  framen;        // frame
    input  [3:0] cben;    // command/byte enable
    inout  [31:0] ad;     // address/data
    input  irdyn;         // initiator ready
    output trdyn;         // target ready
    output devseln;       // device select
    output mem_read_write;          // memory read (1) / write (0)
    input  mem_ready;               // memory ready
    output [31:0] mem_addr;         // memory address
    output [31:0] mem_data_write;   // data to memory
    input  [31:0] mem_data_read;    // data from memory
    output [1:0] state; // for test
    reg pre_framen; // for detecting falling edge of framen
    always @ (posedge clk) begin
        pre_framen <= framen;
    end
```

```
// state transfer
reg  [1:0] next_state = `IDLE; // next state
reg [31:0] auto_addr  = 0;     // address for burst mode
always @(posedge clk or negedge rstn) begin
    if (!rstn) begin
        next_state <= `IDLE;
        auto_addr   <= 0;
    end else begin
        if (!framen && pre_framen) begin
            if (ad[31:16] == 16'hffff) begin
                case (cben)
                    4'b0110: begin next_state <= `R_MEM;
                                   auto_addr  <= ad; end
                    4'b0111: begin next_state <= `W_MEM;
                                   auto_addr  <= ad; end
                    default: begin next_state <= `IDLE;
                                   auto_addr  <=  0; end
                endcase
            end
        end else begin
            case (next_state)
                `R_MEM: begin
                    if (!irdyn && !trdyn) begin
                        auto_addr <= auto_addr + 1;
                    end else begin
                        if (framen && irdyn) begin
                            next_state <= `IDLE;
                        end
                    end
                end
                `W_MEM: begin
                    if (!irdyn && !trdyn) begin
                        auto_addr <= auto_addr + 1;
                    end else begin
                        if (framen && irdyn) begin
                            next_state <= `IDLE;
                        end
                    end
                end
            endcase
        end
    end
end
// memory signals
wire write = (state == `W_MEM);
```

```
    assign mem_read_write = ~(write & ~irdyn & ~trdyn);
    assign mem_data_write = write ? ad : 32'hzzzzzzzz;
    reg  [1:0] state;    // state for memory access
    reg [31:0] mem_addr; // memory address
    always @(negedge clk) begin
        state <= next_state;
        mem_addr <= auto_addr;
    end
    // PCI output signals
    wire enable = (state == `R_MEM) & ~(framen & irdyn);
    assign ad = enable ? mem_data_read : 32'hzzzzzzzz;
    assign trdyn = ~mem_ready;
    assign devseln = ~((state != `IDLE) & ~(framen & irdyn));
endmodule
```

图 15.43 是存储器写操作的仿真波形。写存储器时的起始地址是 FFFFFFF0，连续写三个数据：55550000、55551111 和 55552222。从图中可以隐约看出存储器的地址是自动增 1 的。三个写操作的位置在图中用了三个时间标尺标出。图 15.43 与 PCI 定义的时序完全一致，请与图 15.40 进行比较。

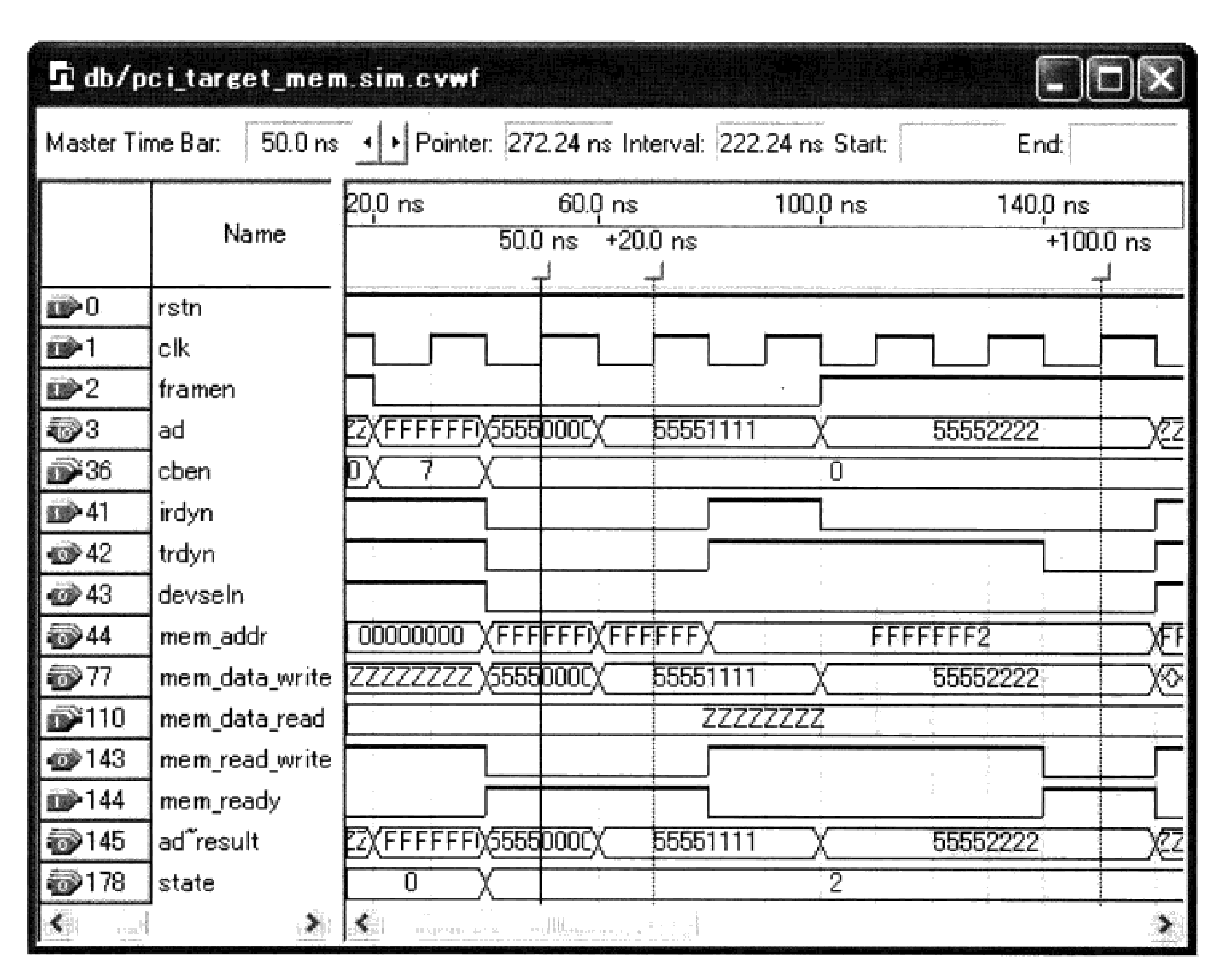


图 15.43　PCI 仿真波形 (存储器写)

图 15.44 是存储器读操作的仿真波形，它与图 15.39 也有相同的时序，也是读出三个数据。该图中的地址自动增 1 显示得比较清楚。另外我们也可以看出，在最后一次读操作完成时，state 仍在下一个周期表示是读状态。但如前所述，我们封锁了 enable，使得 ad 的输出为高阻 (见 ad~result 信号的最右端)。

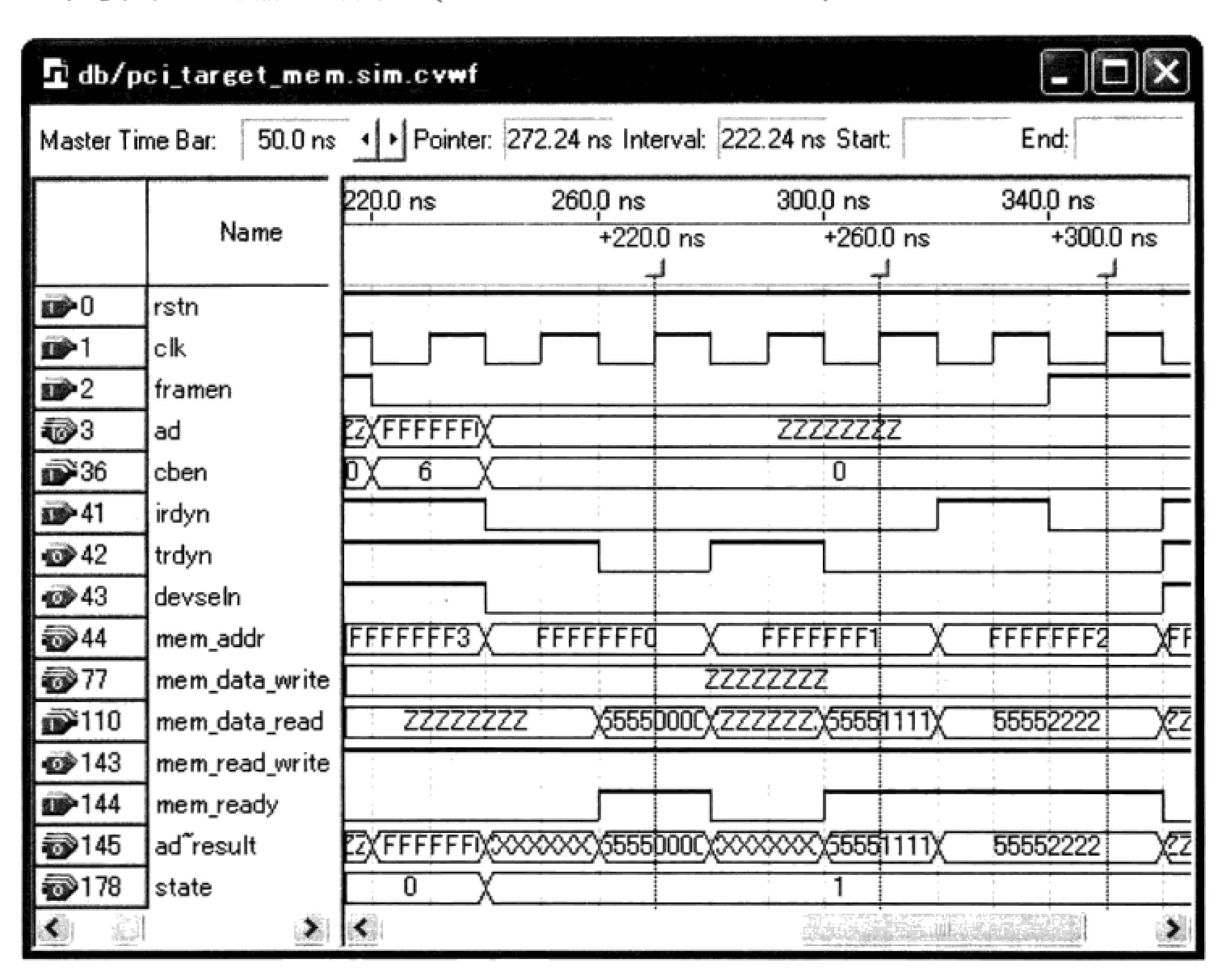


图 15.44 PCI 仿真波形 (存储器读)

15.7 习题

1. 参考图 15.5，试用 Verilog HDL 设计一个 CRC-16-CCITT 的 CRC 码生成电路。
2. 使用本章 UART 的 Verilog HDL 代码作为核心，试设计一个类似于 Intel 8251A 的串行接口控制器。
3. 试设计一个 PS/2 键盘接口控制器，其中包括字符和数字键的键盘扫描码到 ASCII 的转换电路。
4. 试设计一个 PS/2 鼠标控制器，并在屏幕上显示鼠标光标，形状自己决定。
5. 完善书中的 VGA 显示键盘字符的 Verilog HDL 代码，使其能够处理所有键的输入，并加入闪烁的光标。
6. 试设计一个彩色“字符”显示控制器，包括字符缓冲区、字模产生器、VGA 的 RGB、HS 和 VS 信号。

7. 试设计一个彩色汉字显示控制器。一个汉字字模用 16 × 16 点阵表示，显示缓冲区存放汉字的国标码及颜色值。
8. 重做上一题，把彩色字模放入视频存储器。
9. 如果你有一个 FPGA 板并且上面有 I2C EEPROM 或视频输入处理器之类的东西，使用本章的 i2c.v，编写相应的驱动程序，实现与这些器件的通信。
10. 直接存储器访问 (DMA) 控制器 (DMAC) 负责完成 I/O 设备与存储器之间的数据传输。CPU 首先初始化 DMAC，然后 DMAC 向 CPU 发出总线请求。得到响应后，由 DMAC 控制总线完成 I/O 设备与存储器之间的数据传输。试设计一个简单的 DMAC，实现存储器数据块的搬家 (还是往存储器里搬)。
11. 本书给出了一个简单的 PCI 从设备接口电路的例子。试用 Verilog HDL 设计一个主设备的 PCI 接口电路，并实现所有必需的信号。
12. 调查 USB 的接口标准。有可能的话用 Verilog HDL 设计一个 USB 键盘或鼠标的接口电路。

第 16 章　高性能计算机及互联网络设计

多核 CPU 的性能比单核 CPU 的性能有所提高，但如果一个计算机系统中只有一个 CPU 的话，其性能不能满足大规模并行计算的需求。高性能计算机一般是指包含多个 CPU 的计算机系统。多个 CPU 或多个计算机节点由互联网络 (Interconnection Networks) 连接在一起。本章讨论高性能计算机的结构及互联网络的设计。

16.1　高性能计算机的种类

高性能计算机分为两种：并行系统 (Parallel Systems) 和分布式系统 (Distributed Systems)。并行系统的特点是所有的 CPU 共享所有的存储器，即不管存储器藏在什么地方，系统中的任何一个 CPU 都能访问到它。我们又称之为共享存储器的多处理机 (Multiprocessors) 系统。分布式系统由多个计算机组成，每个计算机中的存储器只有该计算机中的 CPU 才能访问。计算机之间的通信由消息传递 (Message Passing) 完成。我们有又之为多计算机 (Multicomputers) 系统。计算机网络系统可以归到此类。

本节重点讨论并行系统。按存储器布局的不同，并行系统又可分为集中式共享存储器 (Centralized Shared Memory) 系统和分布式共享存储器 (Distributed Shared Memory) 系统。

16.1.1　集中式共享存储器系统 (SMP)

集中式共享存储器 (Centralized Shared Memory) 系统的结构见图 16.1。存储器模块可能有一个，也可能有多个，但每个 CPU 在访问存储器时都呈现相同的特点，具有相同的访问时间。我们称这类系统具有 UMA (Uniform Memory Access) 特性。我们又称集中式共享存储器的多处理机为对称型多处理机 (Symmetric Multiprocessors, SMP)。图中的互联网络是总线。

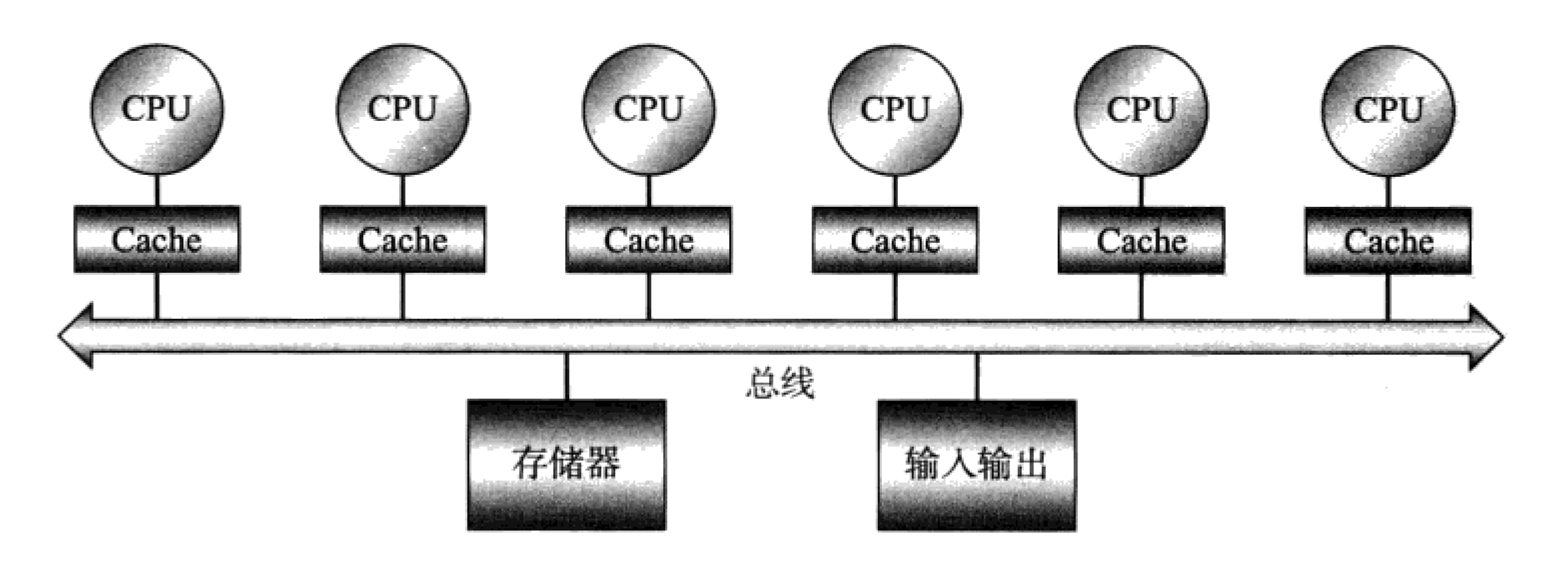


图 16.1　集中式共享存储器多处理机结构图

使用总线的对称型多处理机系统可以连接的 CPU 个数比较少，往往用于构建伺服器 (Servers) 等小规模的并行系统，因为总线会成为系统性能的“瓶颈”。

16.1.2 分布式共享存储器系统 (DSM)

分布式共享存储器 (Distributed Shared Memory，DSM) 系统的结构见图 16.2。存储器是共享的，这没问题，但不是集中的，而是分散在每个 CPU 电路板上。这样，访问存储器时就有两种情况：本地存储器 (Local Memory) 访问和远程存储器 (Remote Memory) 访问。当 CPU 访问自己板上的存储器时，不需打扰互联网络，直接访问就行。而当 CPU 访问其他 CPU 板上的存储器时，需要经过互联网络。与本地存储器访问相比，远程存储器访问要花费更长的时间。我们称这类系统具有 NUMA (Non-Uniform Memory Access) 特性。

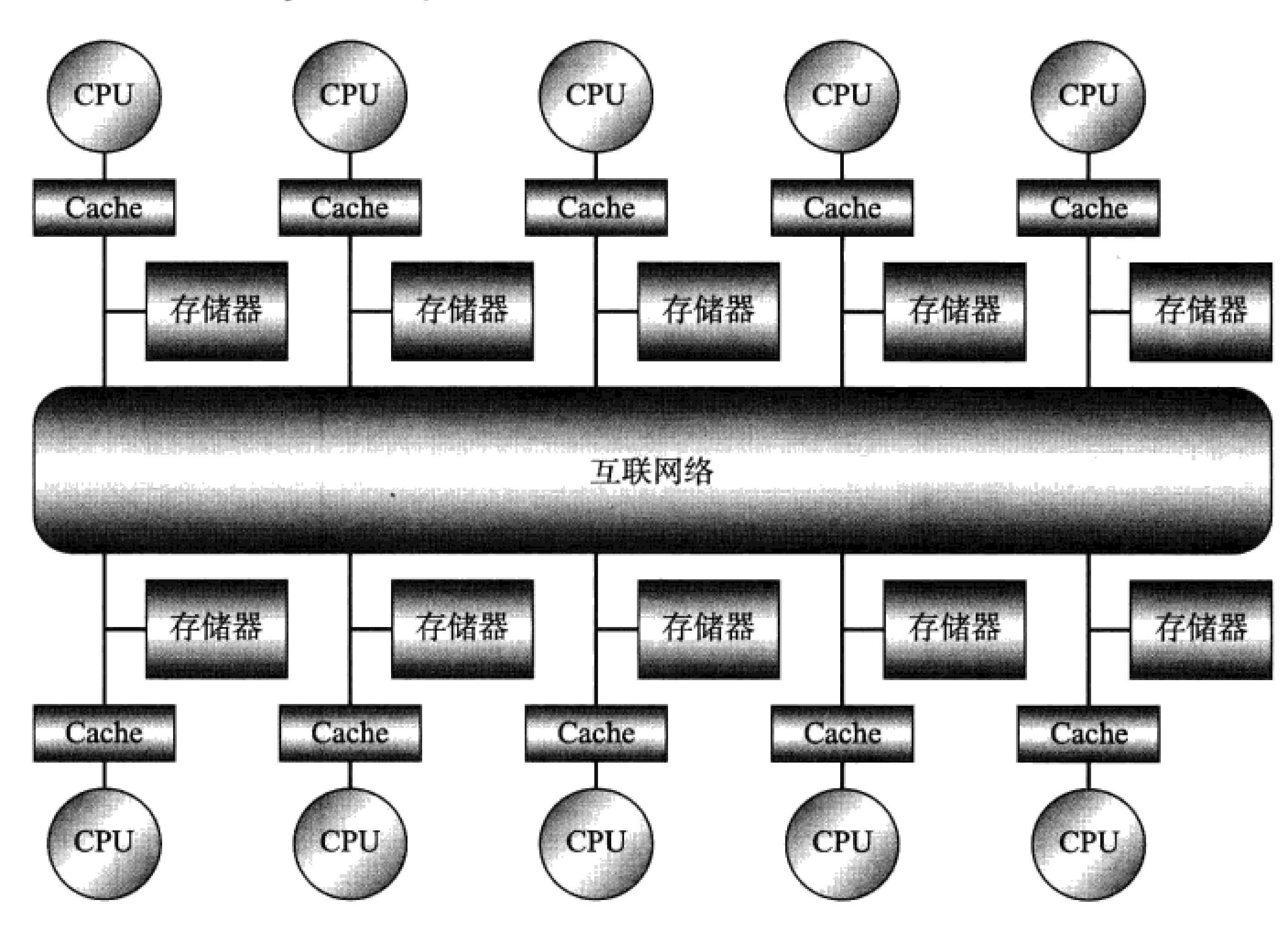


图 16.2 分布式共享存储器多处理机结构图

分布式共享存储器方式可以实现大规模甚至超大规模的并行系统。全球 500 强 (http://www.top500.org) 的超级计算机 (Supercomputers) 大都采用这种方式。

不管是集中式还是分布式，由于存储器为所有 CPU 共享并且每个 CPU 都有自己的 Cache，在这种系统中存在一个著名的 Cache 一致性问题 (见第 14 章对多核 CPU 的 Cache 一致性问题的描述)。在采用总线的集中式共享存储器系统中，解决这个问题的方法是使用总线监视协议；而分布式共享存储器系统使用基于目录的 Cache 一致性协议。后者的基本思想是为每个存储器块建立一个目录，标出在哪些 Cache 中保存有该块的备份。

16.2 互联网络的构成

在并行系统中，互联网络的作用是连接所有的 CPU 和存储器，使它们能够通信。互联网络由两部分组成：一是带有多个通信端口的互联开关 (Switch)，每个 CPU/存储器板都需要一个这样的互联开关；二是链路 (Link)，通常是电缆线 (Cables)，依照某种拓扑结构 (Topology) 连接各互联开关的通信端口，见图 16.3。互联开关与 CPU/存储器板之间的连接端口可以是专用端口，也可以是一般的通信端口。我们把 CPU/存储器板和连接它的互联开关合在一起，称其为节点 (Node)。

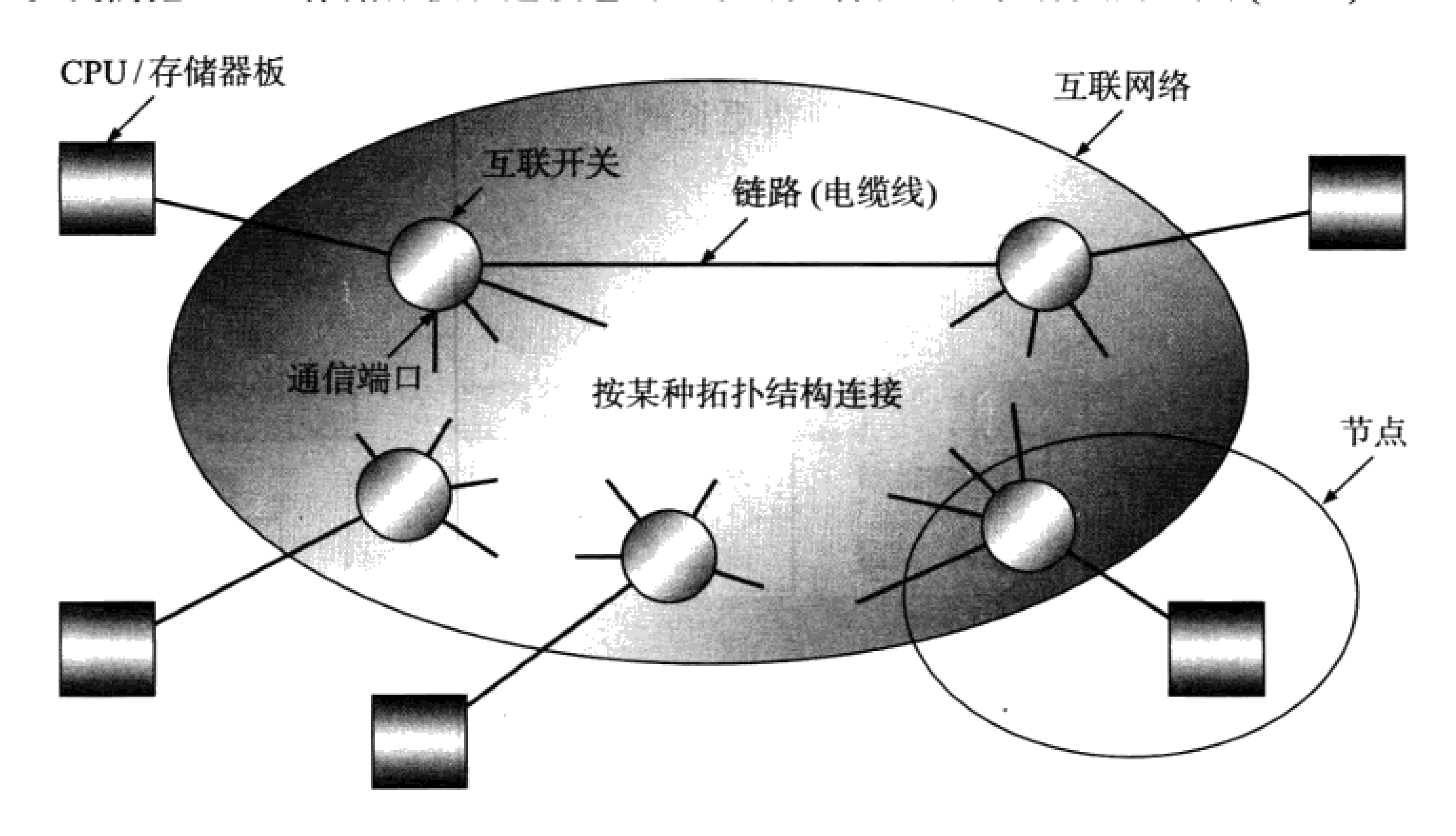


图 16.3 互联网络结构

互联开关有多个双向的通信端口，它们可以是串行的，也可以是并行的。一个 8 串行端口的互联开关的结构如图 16.4 所示。输入端有缓冲区，用来保存到来的数据。交叉开关 (Crossbar) 为每个输出端口选择数据来源。图中的 Port1InputLink (输入) 和 Port1OutputLink (输出) 合在一起，构成 1 号双向通信端口。其余类似。

一个 N × N 的交叉开关有 N 个输入、N 个输出。每个输出都可以从 N 个输入中任选一个送出。一个 32 × 32 交叉开关的结构如图 16.5 所示，Inputs 是 32 个输入，Outputs 是 32 个输出。图中的例子使用 32 × 32 的开关矩阵 (Switch Matrix)。它实际上是由 32 个 32 选 1 的多路器组成的。每个多路器都有一个 5 位的选择信号。该选择信号由配置 (Configuration) 寄存器送出。而配置寄存器数据来自于加载 (Load) 寄存器。加载寄存器也有 32 个，每个有 5 位。当 Load 信号为 1 时，5 位地址 Address 通过译码，选中其中的一个寄存器，把 5 位的 Data 数据在时钟上升沿处写入选中的寄存器中。输入信号 Config 用于把 32 个加载寄存器的内容在时钟上升沿处同时打入配置寄存器。图中的 CS 是片选信号，Reset 是复位信号。复位时，所有的输出端都选择 0 号输入端。

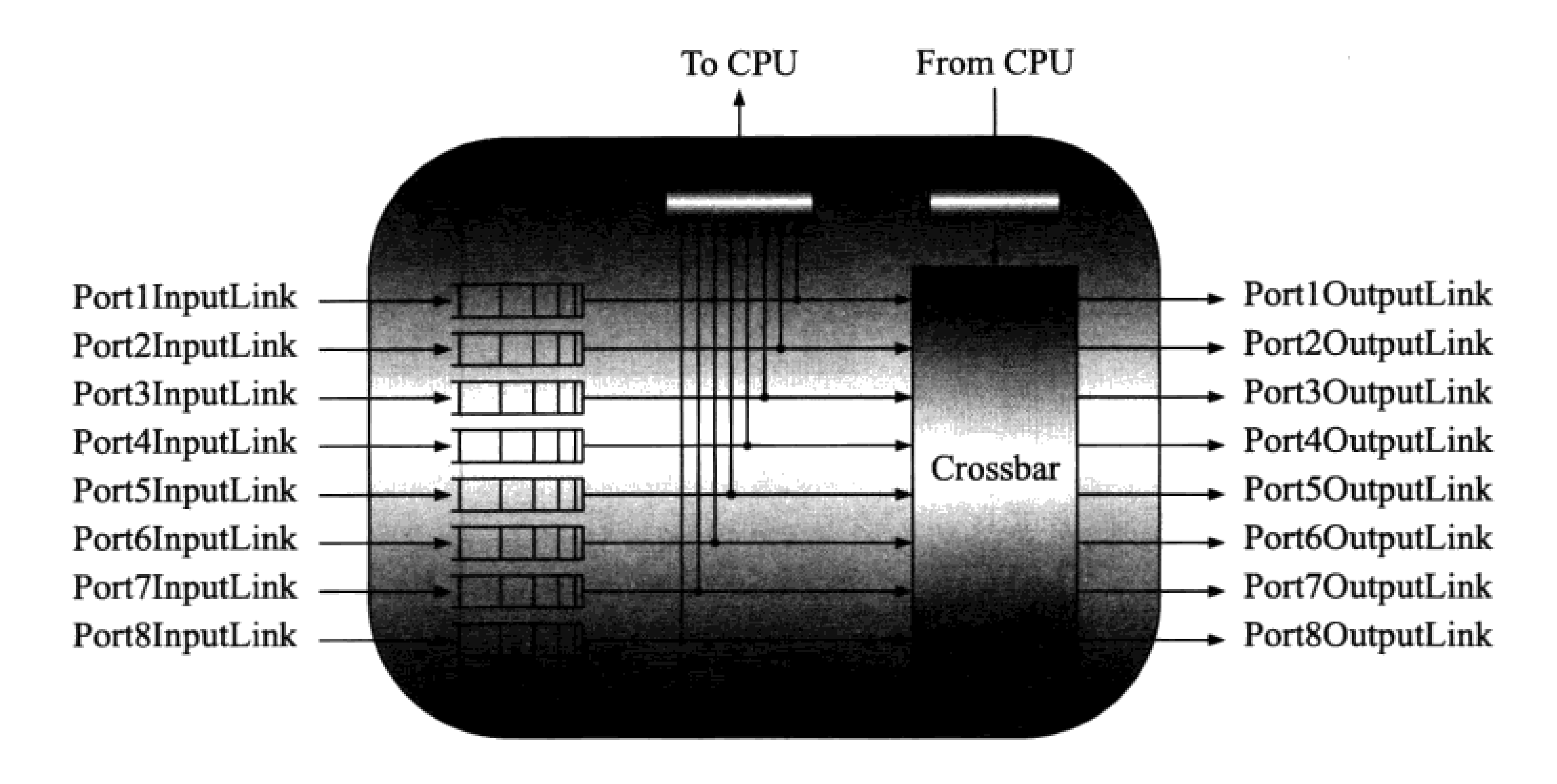


图 16.4 互联网络开关结构图

图 16.5 所示的是一种非常简单的交叉开关电路，多路器的选择信号要一个接一个地顺序写入到 Load 寄存器中，然后由 Config 命令同时修改配置寄存器。这种电路可以实现存储转发 (Store-and-Forward) 式的通信。

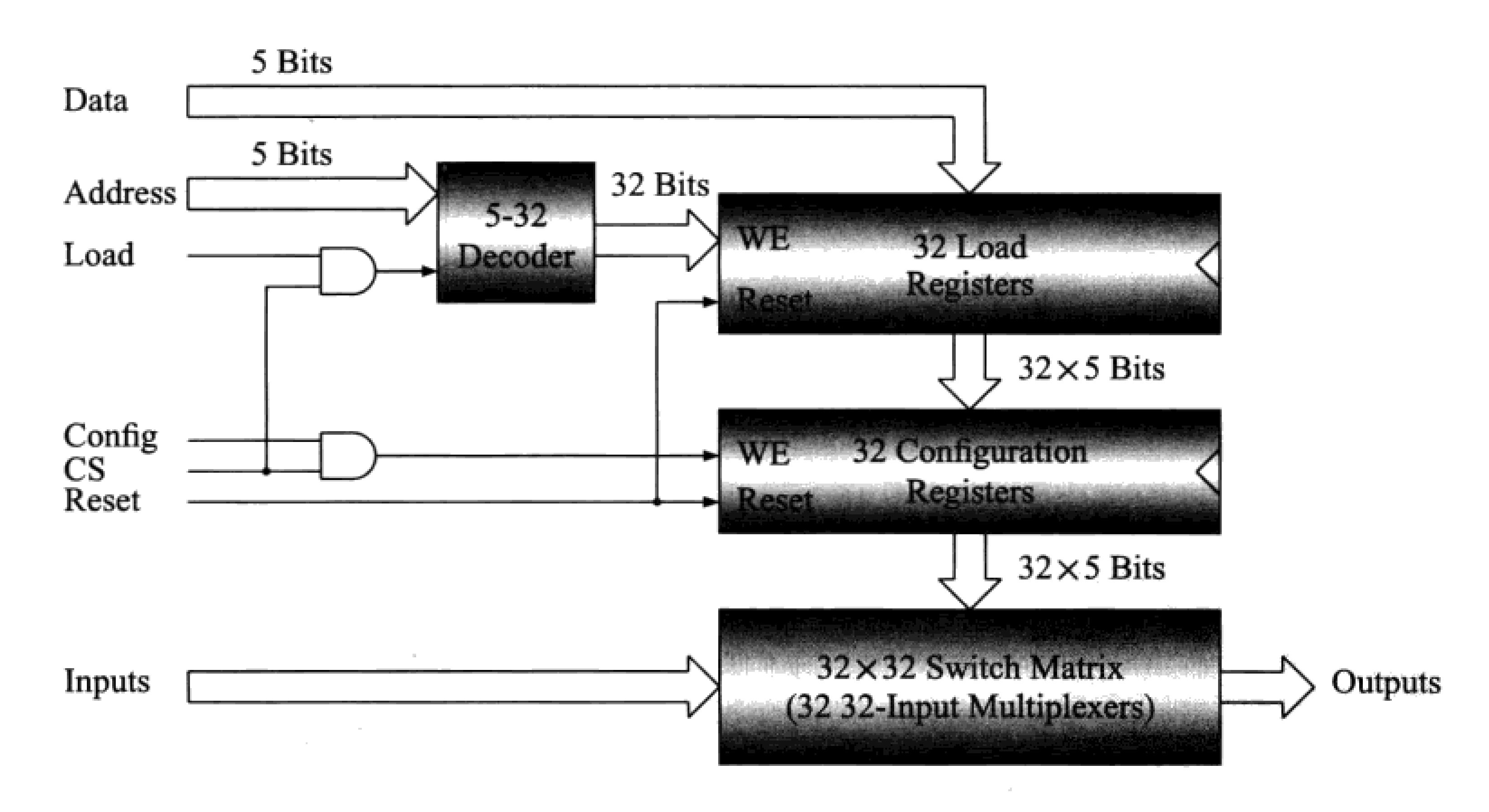


图 16.5 交叉开关电路图

图 16.6 所示的是一种可以根据输入信息中的目的地址 (address) 信息自动产生多路器选择信号的交叉开关电路。多路器的选择信号由输入信息中的地址域自动产生。这种电路可以实现 Cut-Through 式的通信。

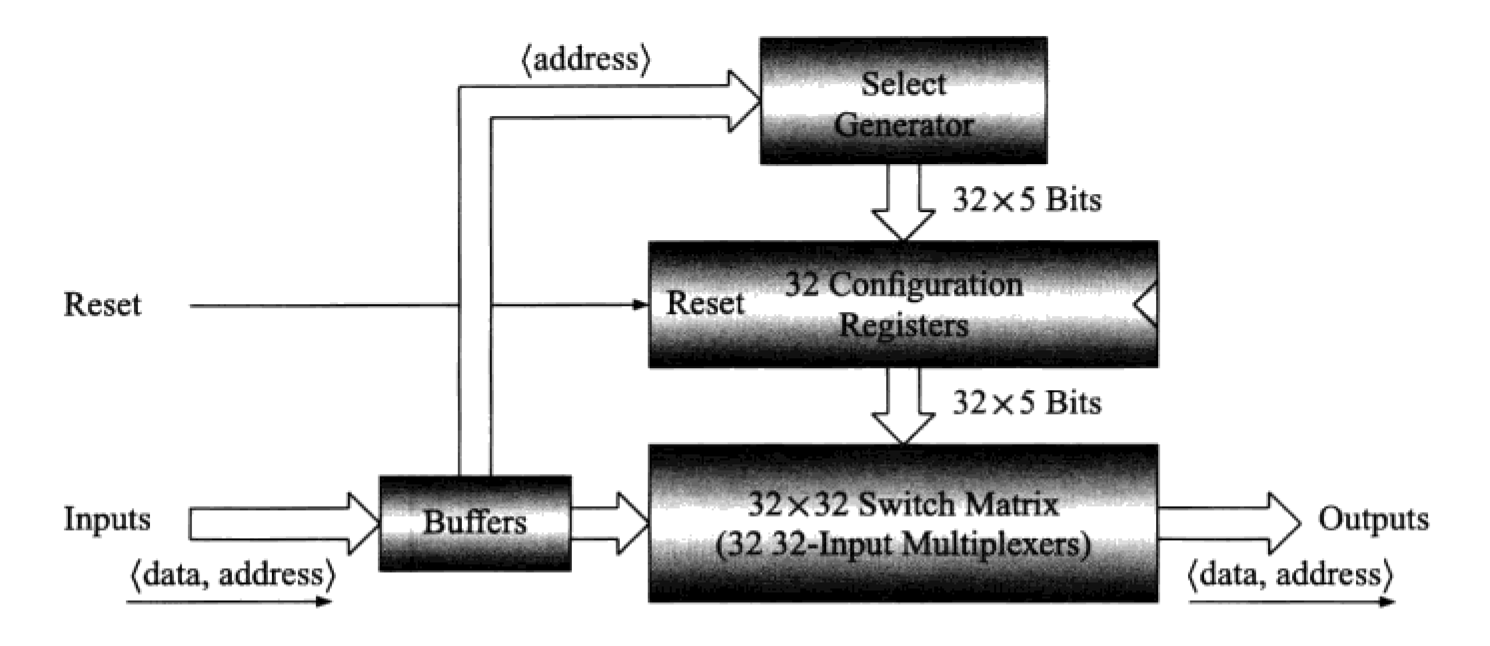
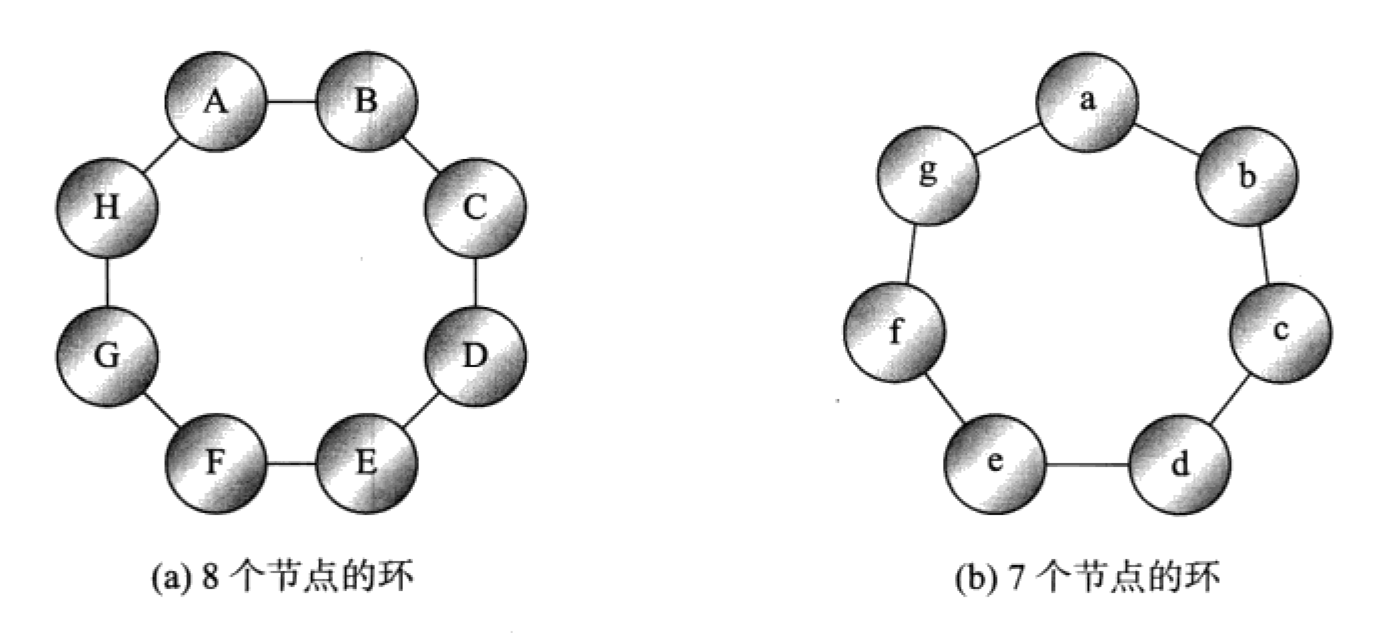


图 16.6 自动路由交叉开关电路图

16.3 互联网络的拓扑特性

我们以环 (Ring) 为例，讨论互联网络固有的拓扑特性 (Topological Properties)。图 16.7(a) 示出了一个有 8 个节点的环，图 16.7(b) 的环有 7 个节点。



(a) 8 个节点的环 (b) 7 个节点的环

图 16.7 环型互联网络

16.3.1 节点度 (Degree)

如果两个节点之间有一条链路，我们称这两个节点互为邻居。我们定义一个节点的度 (Degree) 为该节点的邻居数，也就是一个节点的链路数。例如，图 16.7 中环的度为 2，与节点数量无关。节点度影响互联开关的复杂程度及价格。

16.3.2 直径 (Diameter)

一个互联网络的直径 (Diameter) 定义为任意两个节点之间的最短路径中路径长度的最大值 (链路数)。例如，图 16.7(b) 中环的直径为 3。一般地，如果一个环有 N 个节点，它的直径为 $\lfloor N/2 \rfloor$。互联网络的直径影响通信时间：直径越长，所需通信

时间也越长。我们总是希望有这样一种互联网络：它有比较小的节点度 (设计简单价格低) 并且直径也短 (通信时间短速度快)。环的节点度小但直径长，全连接 (任意两个节点之间都有一条链路) 直径短但节点度大。

16.3.3 平均距离 (Average Distance)

两个节点之间的距离是它们之间的最短路径的长度，也就是链路的数量。例如，图 16.7(a) 中节点 A 与节点 F 之间的距离是 3。一个互联网络的平均距离定义如下：所有的节点对 (Pairs) 中的两个节点之间的距离的总和除以对数 (包括一个节点和它本身)。

如果互联网络是对称的，则以上计算可以简化：假设共有 N 个节点，任取一个节点，计算该节点到所有节点距离的总和 (一个节点到它本身的距离为 0)，再除以 N。例如，环是对称的互联网络，对图 16.7(a)，取节点 A 来计算平均距离：节点 A 到节点 A、B、C、D、E、F、G 和 H 的距离分别为 0、1、2、3、4、3、2 和 1。平均距离为 $(0+1+2+3+4+3+2+1)/8=2$。

16.3.4 对分宽度 (Bisection Bandwidth)

试着搞一下破坏：用钳子 (剪刀可能也行) 剪断最少数量的链路 (电缆线)，使所有节点分成数量相等 (或差 1 个) 的两部分：两部分之间的链路全被剪断了。这个被剪断的链路的数量就是对分宽度 (Bisection Bandwidth)。注意：你可别真剪，在纸上比划比划就行了。图 16.7 中环的对分宽度为 2，与节点数量无关。

16.4 常用的互联网络

以下介绍几种简单且常用的互联网络的拓扑结构并给出它们的拓扑特性。它们是 Mesh、Torus、Hypercube 和 Tree[1]。

16.4.1 Mesh

图 16.8(a) 和图 16.8(b) 示出了二维 (2D) Mesh 和三维 (3D) Mesh 两个例子。Mesh 不是对称的互联网络，处在边角的节点的度与处在中心位置的节点的度是不同的。

16.4.2 Torus

给 Mesh 中的边角节点加入额外的链路，使所有节点具有相同的度，就变成了 Torus。图 16.9(a) 和图 16.9(b) 示出了二维 (2D) Torus 和三维 (3D) Torus 两个例子。很多高性能计算机都采用 3D Torus 互联网络。它具有固定的节点度 6，而且是对称的。但当系统有相当多的节点时，3D Torus 的直径比较长。例如，如果节点数为 $128\times128\times64=1\,048\,576$，它的直径为 $64+64+32=160$。

[1] Tree 是树，Hypercube 是超立方体，Mesh 和 Torus 不好翻译成中文，网格和环格？好像不怎么好。如同 Cache 一样，干脆全部直接用英文名称好了。

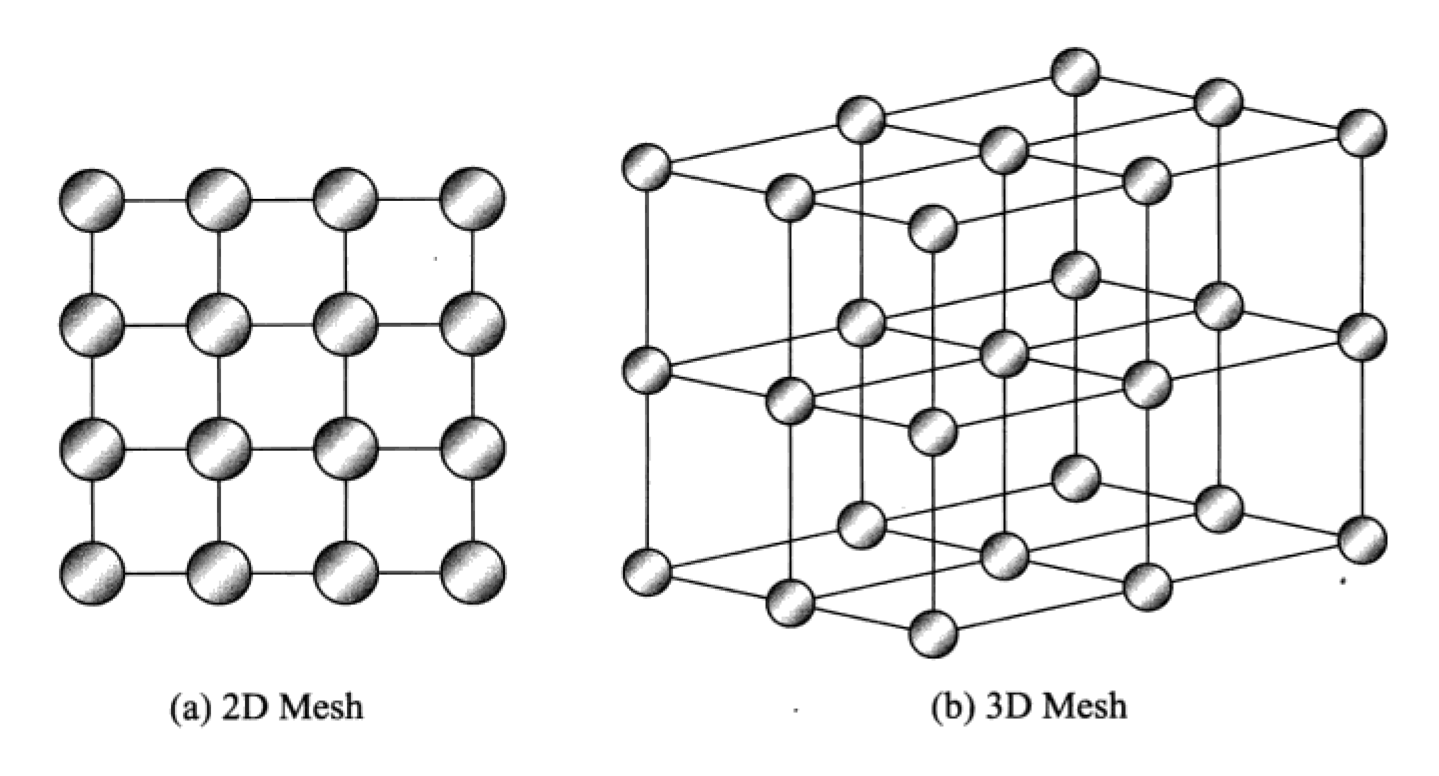


图 16.8 Mesh 的拓扑结构

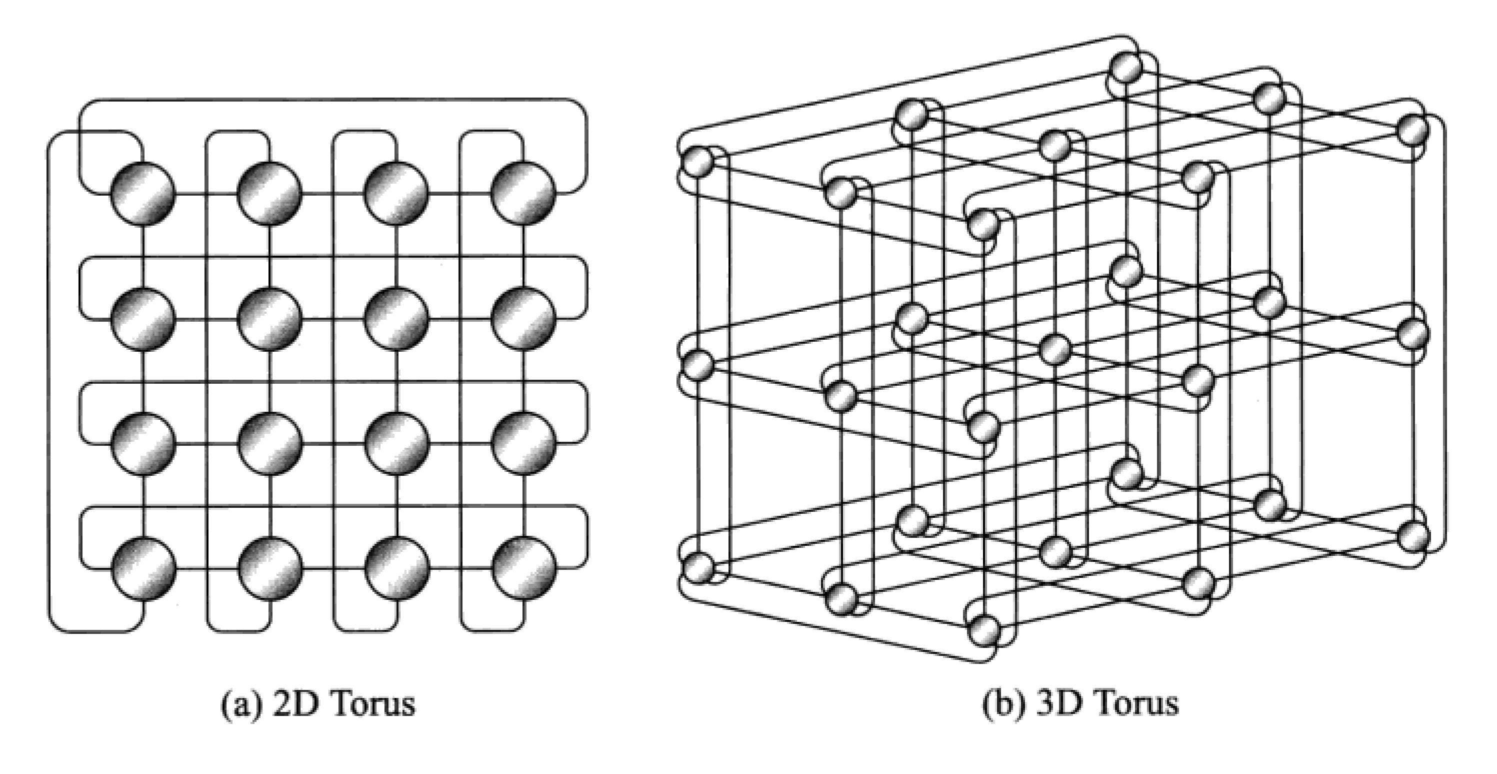


图 16.9 Torus 的拓扑结构

16.4.3 Hypercube

Hypercube (超立方体) 是一种非常有意思的且常用的互联网络，它具有递归的特性。图 16.10 示出的是 n 维 Hypercube ($n = 0, 1, 2, 3, 4, 5$)。一般地，n 维 Hypercube 也称为 n-cube。一个 n-cube 有 2^n 个节点，节点度为 n，直径也为 n。例如，如果节点数为 $2^{20} = 1\,048\,576$，20-cube 的直径为 20，比 3D Torus 小很多，但节点度比 3D Torus 大不少。

16.4.4 Tree 和 Fat-Tree

Tree (树) 也是一种常用的互联网络。图 16.11(a) 示出的是一个具有 15 个节点的 Binary Tree (二叉树)。我们称节点 r 为根节点 (树根在上面，长倒了)。它有两个子

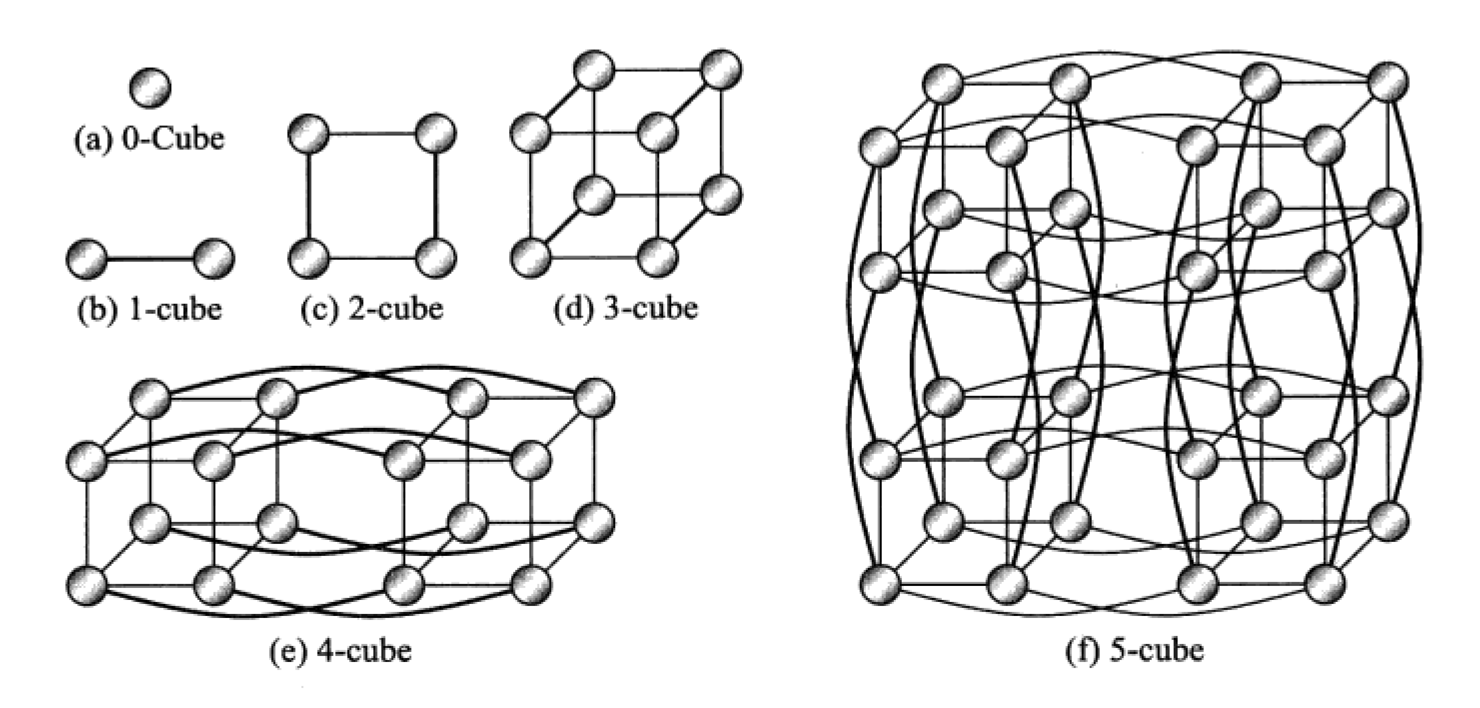


图 16.10 Hypercube 的拓扑结构

树：根节点为 s 的左子树和根节点为 t 的右子树 (称为左父右母似乎更合适)。

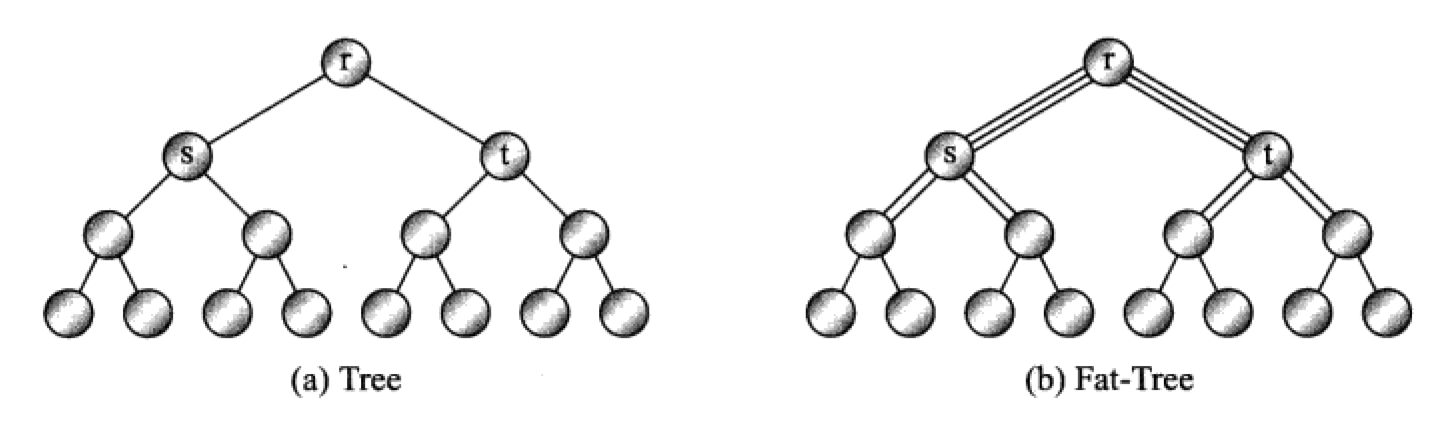


图 16.11 Tree 和 Fat-Tree 的拓扑结构

由于左子树的节点与右子树的节点通信时都要经过根节点 r，链路 (s, r) 和 (t, r) 将成为通信的瓶颈。为了解决这个问题，我们在瓶颈处加入额外的链路，就构成所谓的 Fat-Tree (树枝比较细、树干比较粗的树)，如图 16.11(b) 所示。

16.5 基本的通信操作

本节以 Hypercube 为例，讨论基本的通信操作。我们采用存储转发方式实现通信，即一个节点接收完一个完整的信息包后，再向下一个节点发送。向一个相邻节点发送一个信息包的时间为 $t_s + mt_w$，其中 t_s 为信息包的准备时间，m 为信息包的字数，t_w 为发送一个字所需的时间。

Hypercube 中每个节点的地址的指定方法如图 16.12 所示。我们使用一个 n 位的二进制数来表示一个节点的地址。当两个节点的地址只有一位不同时，这两个节点之间有一条链路。

假设源节点 s 想要给目的节点 d 发送信息，我们可以从节点地址的最高位开始向最低位逐位检查，如果两个地址位不同，则发送。图 16.13 示出的是源节点 010 发

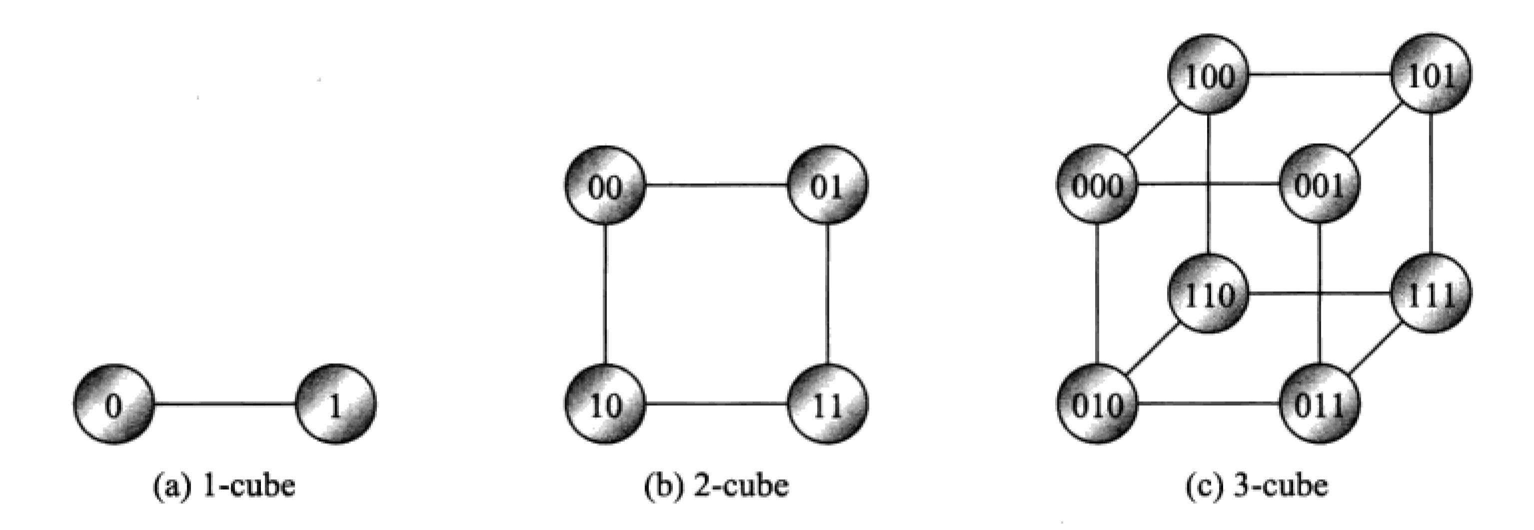


图 16.12 Hypercube 节点地址

送信息给目的节点 101 的过程：010 发送到 110，然后发送给 100，最后发送至目的节点 101。路径为 010 → 110 → 100 → 101。当然，我们也可以从最低位开始检查，如此得到不同的发送路径：010 → 011 → 001 → 101。

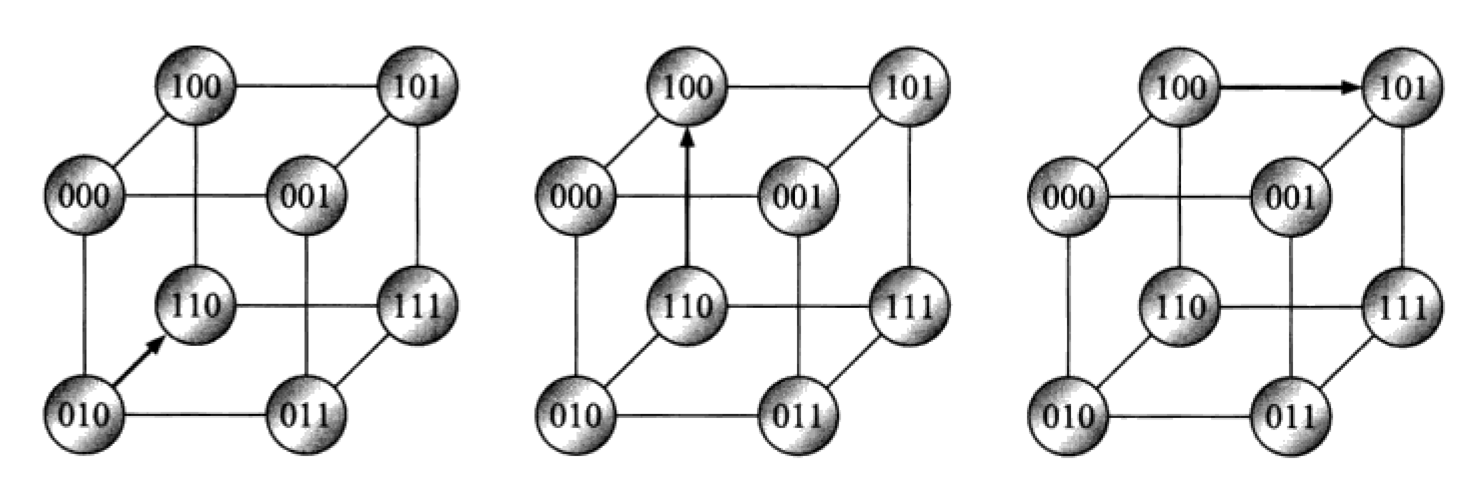


图 16.13 Hypercube 一对一通信

基本的通信操作包括以下四种：(1) 一对多广播 (One-to-All Broadcast)；(2) 多对多广播 (All-to-All Broadcast)；(3) 一对多私通 (One-to-All Personalized Communication)；(4) 多对多私通 (All-to-All Personalized Communication)。最后一种也称做全交换 (Total Exchange)。以下我们以 Hypercube 为例，描述这四种操作和它们所花的时间。设 p 为 n-cube 的节点数，则 $p = 2^n$。我们也假设一个节点在同一时刻只能有一个发送和一个接收。

16.5.1 一对多广播

一对多广播是指一个节点 (源节点) 发送相同的信息给其他所有的节点。我们仍以从高到低的次序逐位发送。首先，沿第 $n-1$ 位的方向，节点 s 发送信息给节点 $s^{(n-1)}$ (s 和 $s^{(n-1)}$ 的地址只是在第 $n-1$ 位处不同)。这时有两个节点保有信息。然后，沿第 $n-2$ 位的方向，节点 s 和 $s^{(n-1)}$ 同时分别发送信息给节点 $s^{(n-2)}$ 和 $s^{(n-1)(n-2)}$。这时有四个节点保有信息。其次，这四个节点再沿第 $n-3$ 位的方向发送信息。这时就有八个节点保有信息。如此这般，直到沿第 0 位发送完为止。

图 16.14 是在 3-cube 中实现一对多广播的情况 (s 为节点 000)。不难看出，一对多广播发送所花的时间为 $t_{oab} = (\log_2 p)(t_s + mt_w)$。

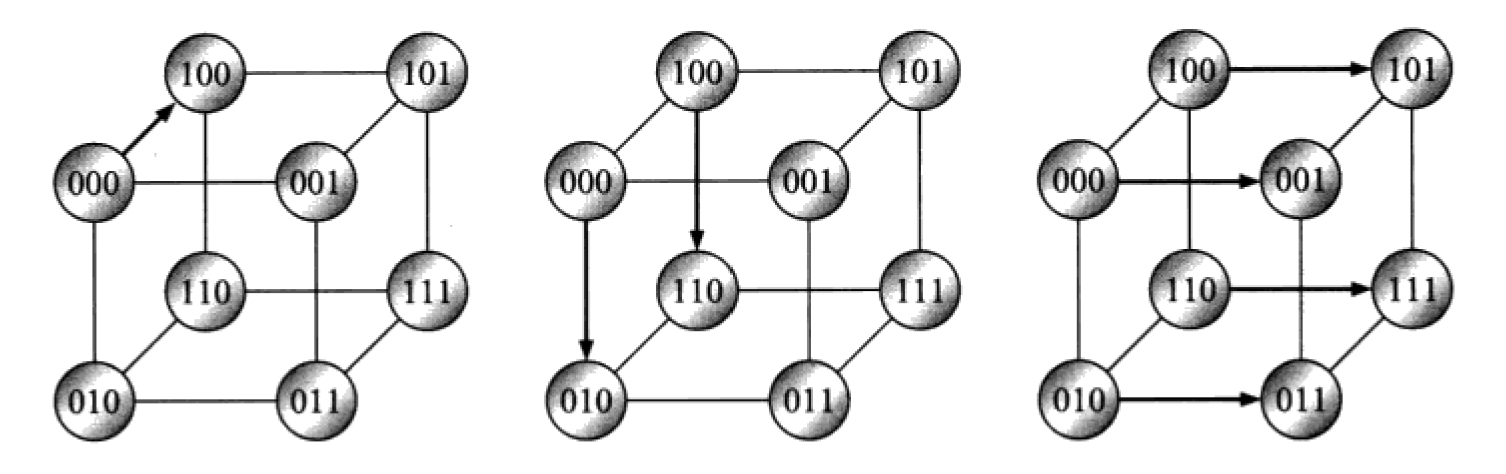


图 16.14 Hypercube 一对多广播

16.5.2 多对多广播

多对多广播是指所有节点都向其他节点广播信息。其实现方法参考图 16.15。图中的 (0,1) 表示节点 0 和节点 1 的信息，其余类推。我们沿第 0、1、2 位的次序交换信息。每次交换都要重新打包，其长度变成原来的两倍。因此多对多广播发送所花的时间为 $t_{aab} = \sum_{i=0}^{n-1}(t_s + 2^i mt_w) = (\log_2 p)t_s + (p-1)mt_w$。

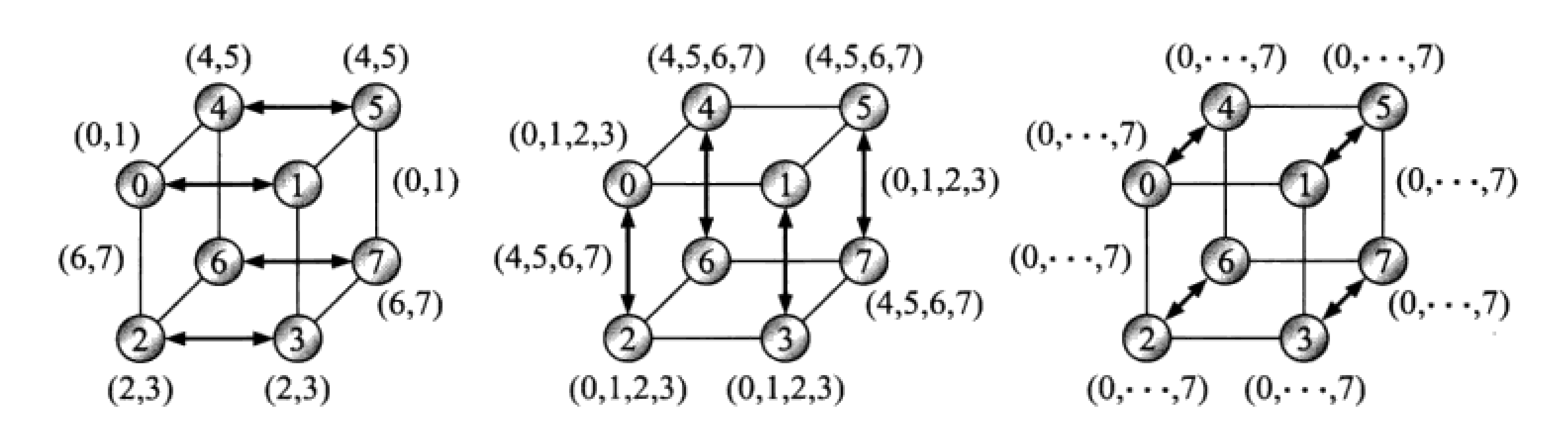


图 16.15 Hypercube 多对多广播

16.5.3 一对多私通

此私通的意思是私人通信 (Personalized Communication)。一对多私通是一个节点发送信息给所有其他节点，与广播不同，每个信息都是不一样的。不失一般性，我们假定节点 0 是源节点，它有 2^n 个信息 (i)，$i = 0, 1, 2, \cdots, 2^n - 1$ (假设也有一个送给它自己的信息)。首先把信息 $(2^{n-1}, \cdots, 2^n - 1)$ 打包，沿最高位第 $n-1$ 位送给节点 2^{n-1}。然后节点 0 和节点 2^{n-1} 再把各自的后一半信息打包，沿第 $n-2$ 位送出。以此类推，每次信息包的长度变为原来的一半，直到沿第 0 位送出为止。

图 16.16 是在 3-cube 中实现一对多私通的情况。所花的时间为 $t_{oap} = \sum_{i=0}^{n-1}(t_s + 2^{n-1-i}mt_w) = (\log_2 p)t_s + (p-1)mt_w$，与多对多广播所花的时间相同。

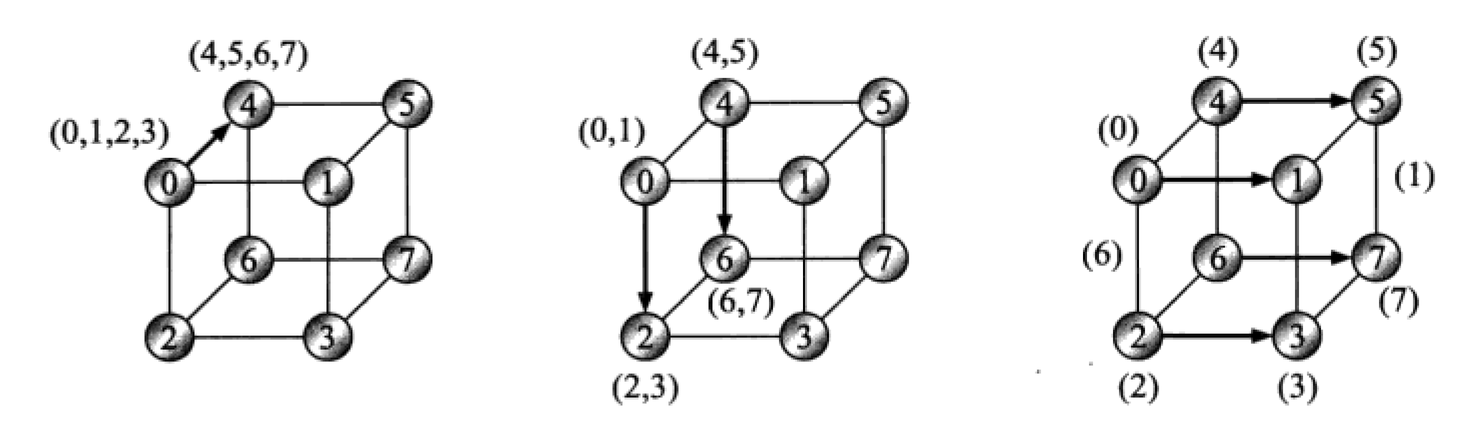


图 16.16 Hypercube 一对多私通

16.5.4 多对多私通

多对多私通是每个节点都发送信息给所有其他节点，并且所有信息都不同。我们假定节点 i 有 2^n 个信息 (i, j)，$i, j = 0, 1, 2, \cdots, 2^n - 1$，i 是信息的源节点号、j 是信息的目的节点号 (假设也有一个送给它自己的信息)。参阅图 16.17，把应该发送的信息打包，按第 $n-1$ 位到第 0 位的次序送出。每次发送的信息包有 $p/2$ 个信息，因此多对多私通所花的时间为 $t_{aap} = (\log_2 p)(t_s + pmt_w/2)$。

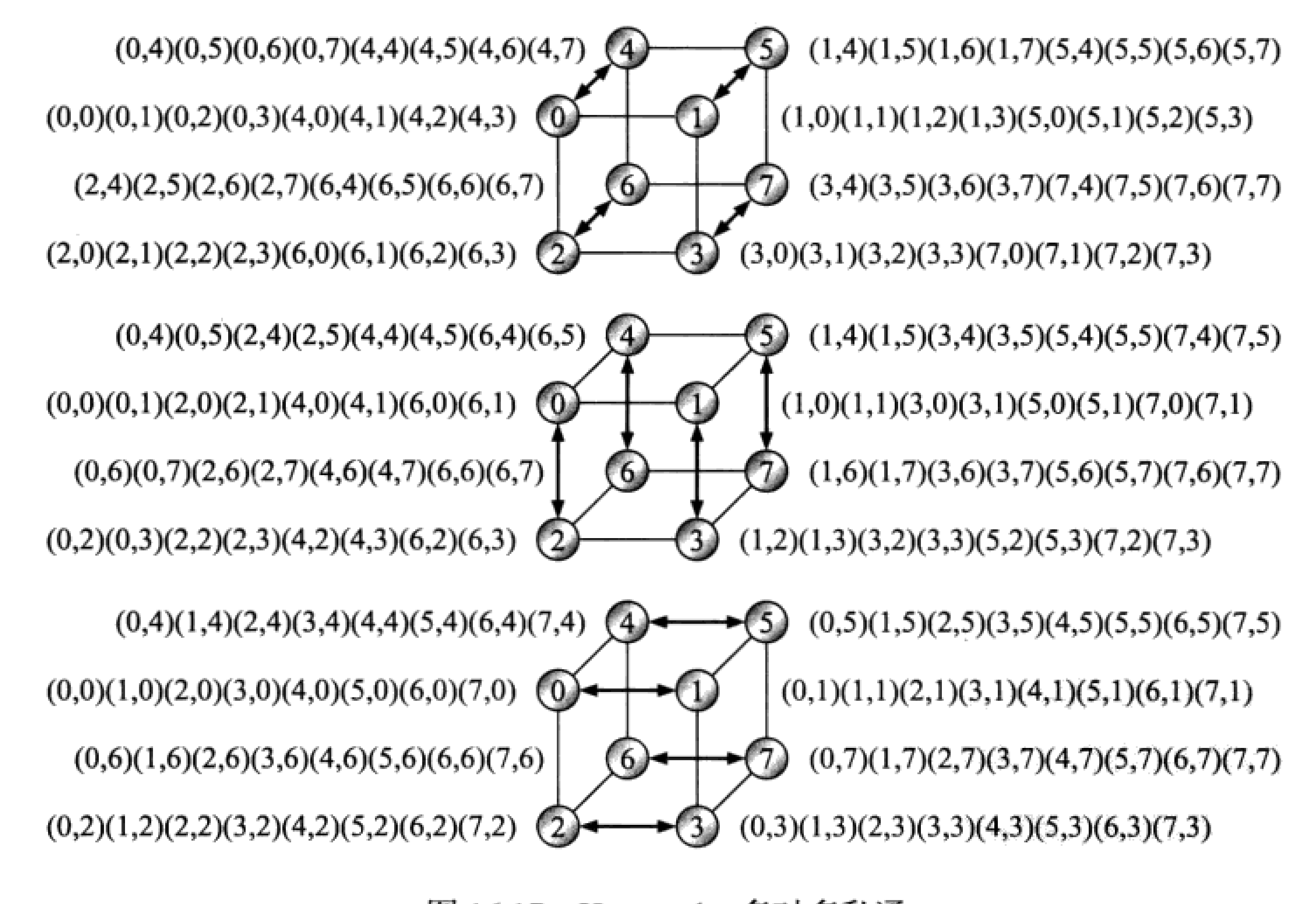


图 16.17 Hypercube 多对多私通

16.6 新型互联网络

我们已经在第二节讲到互联网络由互联开关和连接互联开关的端口的电缆线组成。互联开关的端口越多，其价格也越贵。由于一个节点就需要一个互联开关，当

系统中有成千上万甚至百万个节点时，互联开关的价格往往占了整个系统价格的很大部分。另外，由于要提供高速通信能力，电缆线的价格也不便宜。因此，我们希望在构建大规模的高性能计算机时，对互联开关端口的数量要有所限制。也就是说我们希望使用节点度低的互联网络。

那岂不是说用环型互联网络就好了？问题是环型互联网络的直径太大，比如系统有一百万个节点，那么它的直径就是五十万。为了缩短通信时间，我们又希望互联网络有较小的直径。我们知道，全连接的互联网络(每两个节点之间都有一条链路)的直径最小，但一百万个节点的系统要求互联开关有九十九万九千九百九十九个端口。至少到目前为止，还没有一家公司生产这种产品。

既要互联开关的端口数量比较少，又要互联网络的直径比较小，这是矛盾的呀，是不是要求太高了？是。但我们要朝这个方向努力。本节将要描述的三种新型互联网络 (Dual-Cube、Metacube 和 RDN) 就是这种努力的部分结果。

16.6.1 Dual-Cube

在讨论 Dual-Cube 之前，让我们先熟悉一个比较有名的多处理机系统：SGI 公司的 Origin2000。图 16.18 是 Origin2000 3D 和 4D Hypercube 的系统结构。Origin2000 使用 Hypercube 作为它的互联网络。图中的圆圈表示互联开关，正方形表示 CPU/存储器电路板。每个开关有六个端口，其中两个用来连接 CPU/存储器板，剩下的四个用来构建 Hypercube。图 16.18(b) 使用了互联开关的所有六个端口。

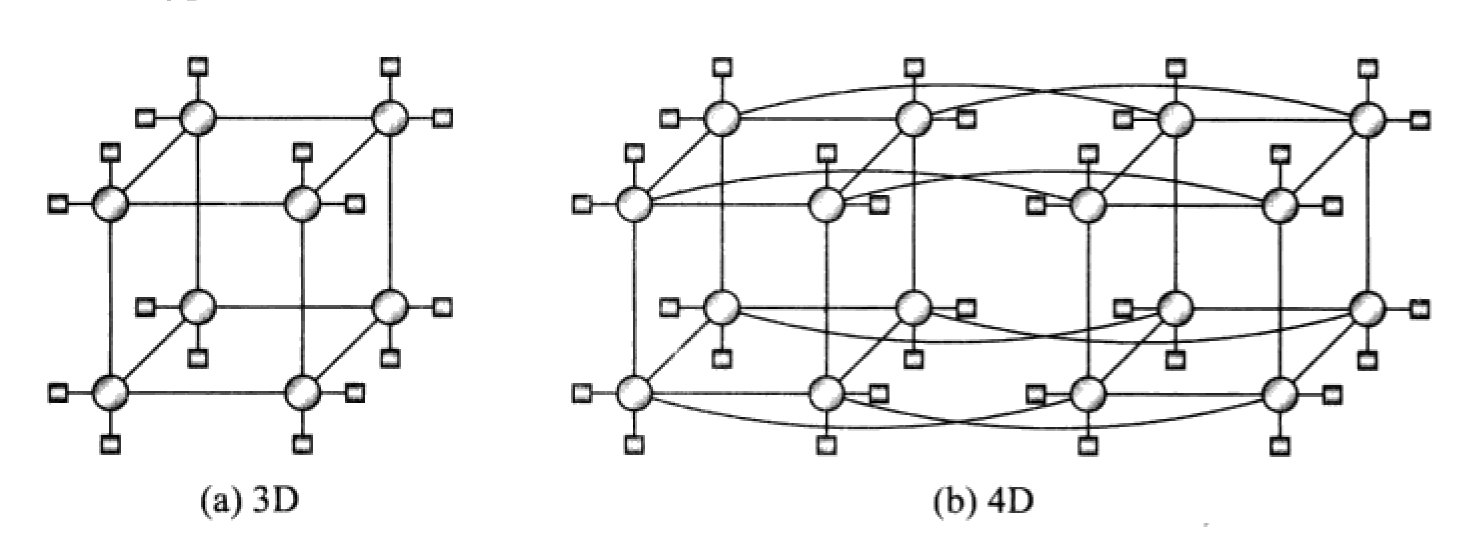


图 16.18 Origin2000 3D 和 4D 互联网络系统结构

如果还想构建一个比图 16.18(b) 规模更大的系统怎么办呢？简单地按 Hypercube 规则增加到 5D 是不可能了，因为已经没有端口可用了。SGI 的做法是使用一个叫做 Cray Router 的互联开关按星型 (Star) 结构连接相应位置的节点，见图 16.19。

当系统规模变大时，互联网络的拓扑结构突然变了。我们不能说这是一种好的做法，因为以前在 Hypercube 上开发的算法和软件都要修改以适应新的拓扑结构。

Dual-Cube[15] 能用较少的链路连接较多的节点，同时尽量保持了 Hypercube 固有的特性。图 16.20 示出 Dual-Cube 的地址格式。一个 Dual-Cube DC(m) 使用 $2m+1$

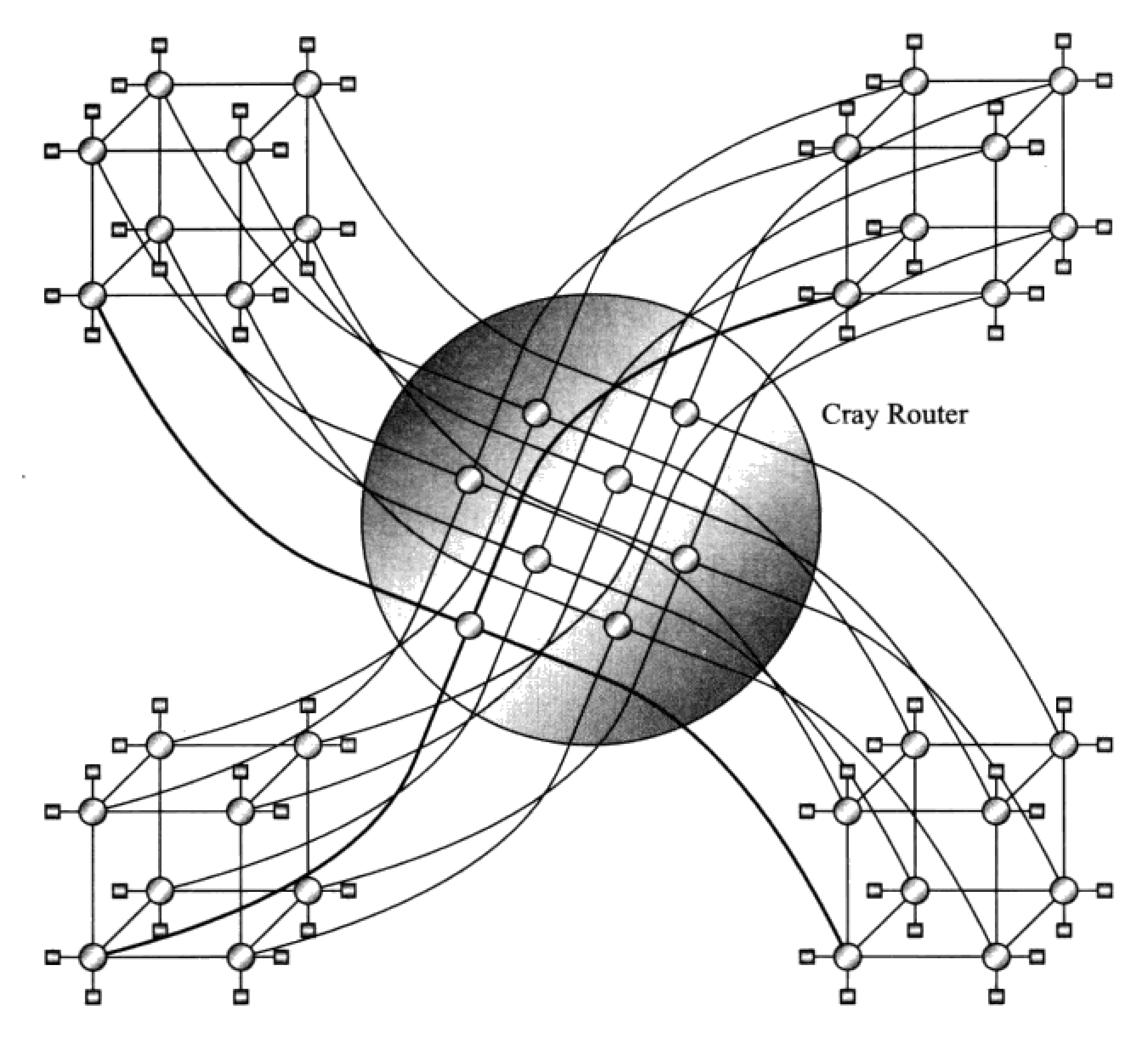


图 16.19　使用 Cray Router 的 Origin2000 5D 互联网络系统结构

位二进制地址，其中的 1 位表示 Class ID，两组 m 位二进制数分别表示 Cluster ID 和 Node ID。注意，在不同的 Class 中，Cluster ID 和 Node ID 所处的位置是不同的。

	1-bit Class ID	m bits	m bits
Class 0	0	Cluster ID	Node ID
Class 1	1	Node ID	Cluster ID

图 16.20　Dual-Cube 地址格式

DC(m) 的节点度为 m + 1。如果两个节点的 2m + 1 位地址只是 Class ID 不同，则两个节点之间有一条链路。另外，如果两个节点的地址只是在 Node ID 域中有一位不同而其他位均相同，两个节点之间也有一条链路。

Dual-Cube 的 Cluster 是一个 m-cube，不同 Cluster 中的两个节点的通信必须要通过 Class ID 域的链路进行。我们称这个链路为 Cross-Edge (m-cube 中的链路为 Cube-Edge)。

图 16.21 示出的是 m = 2 的 Dual-Cube 的结构。每个节点有 3 条链路，总节点数是 32。而传统的 Hypercube 只能连接 8 个节点，如果节点度是 3 的话。图 16.22 示

出的是 DC(3) 的结构。每个节点有 4 条链路，总节点数是 128。图中只画出了 Class 1 的 Cluster ID 为 0 的 8 个节点连接到其他节点的链路。

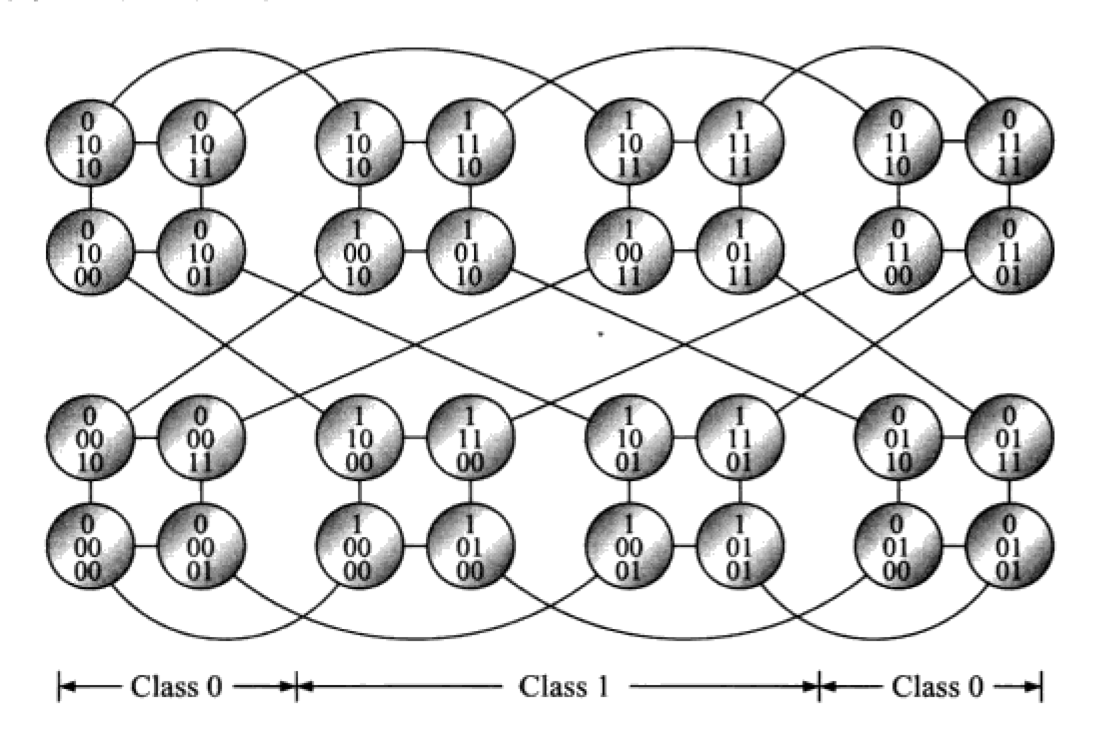


图 16.21 Dual-Cube DC(2) 结构图

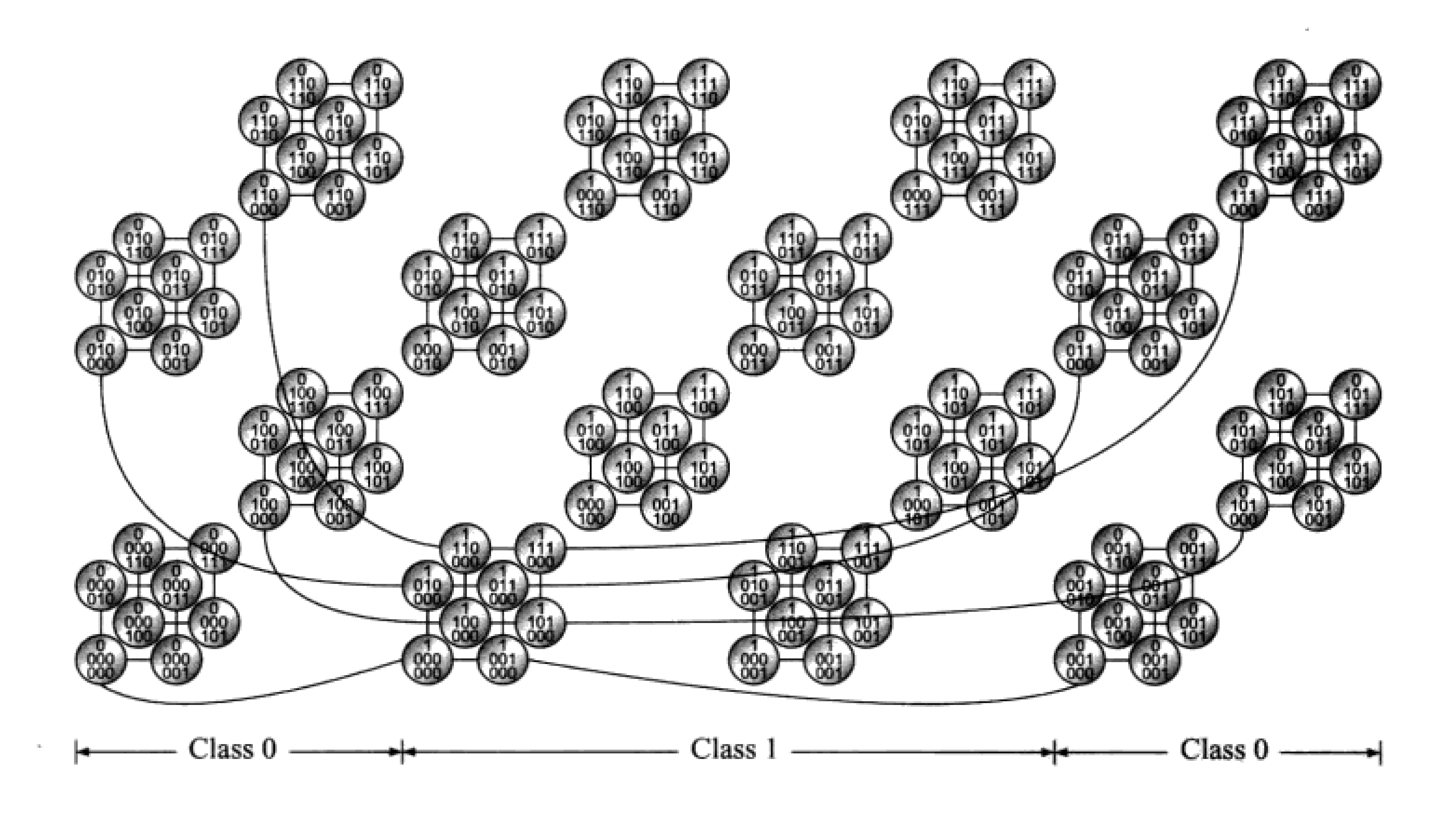


图 16.22 Dual-Cube DC(3) 结构图

在 Dual-Cube 中，建立两个节点之间的通信路径分为以下三种情况，见表 16.1 给出的三个例子。第一种情况是两个节点处在同一个 Cluster (m-cube) 中。这种情况最简单，与 Hypercube 的通信没有任何区别。第二种情况是两个节点的 Class ID 不同。由于 Class ID 不同，这两个节点肯定处在不同的 Cluster 中。这种情况也简单：两个节点各自在自己的 m-cube 中建立路径，使其能通过 Cross-Edge 连起来。第三种情况是两个节点的 Class ID 相同，但处在不同的 Cluster 中。在这种情况下，可先按

表 16.1 Dual-Cube 三种情况下的通信

(1) 相同的 Cluster			(2) 不同的 Class			(3) 相同的 Class 不同的 Cluster		
s	=	0 0000 0000	s	=	0 0000 0000	s	=	0 0000 0000
		0 0000 0001			0 0000 0001			0 0000 0001
		0 0000 0011			0 0000 0011			0 0000 0011
		0 0000 0111			0 0000 0111			0 0000 0111
t	=	0 0000 1111			0 0000 1111			0 0000 1111
					1 0000 1111			1 0000 1111
					1 0001 1111			1 0001 1111
					1 0011 1111			1 0011 1111
					1 0111 1111			1 0111 1111
			t	=	1 1111 1111			1 1111 1111
						t	=	0 1111 1111

第二种情况处理，再走一次 Cross-Edge 就行了。第三种情况告诉我们 Dual-Cube 的直径是 $2m + 2$，比 Hypercube 只多1 ($n = 2m + 1$)。

一般地，如果节点度为 $m + 1$，传统的 Hypercube 能连接 2^{m+1} 个节点，而 Dual-Cube 能连接 2^{2m+1} 个节点，是 Hypercube 的 2^m 倍，代价是直径增 1 (在二者具有相同节点数的情况下)。

回到 Origin2000 的例子。每个互联开关有 6 个端口，两个用来连接 CPU / 存储器板，剩下的 4 个用于构建互联网络。如果使用 Dual-Cube，则 $m = 3$，可连接 $2^7 = 128$ 个节点，而且不用 Cray Router，见图 16.23。图中只画出了部分链路，它们实际上是电缆线，按 Dual-Cube 拓扑结构连上就行，而不需要改变 CPU / 存储器板上的任何硬件电路。

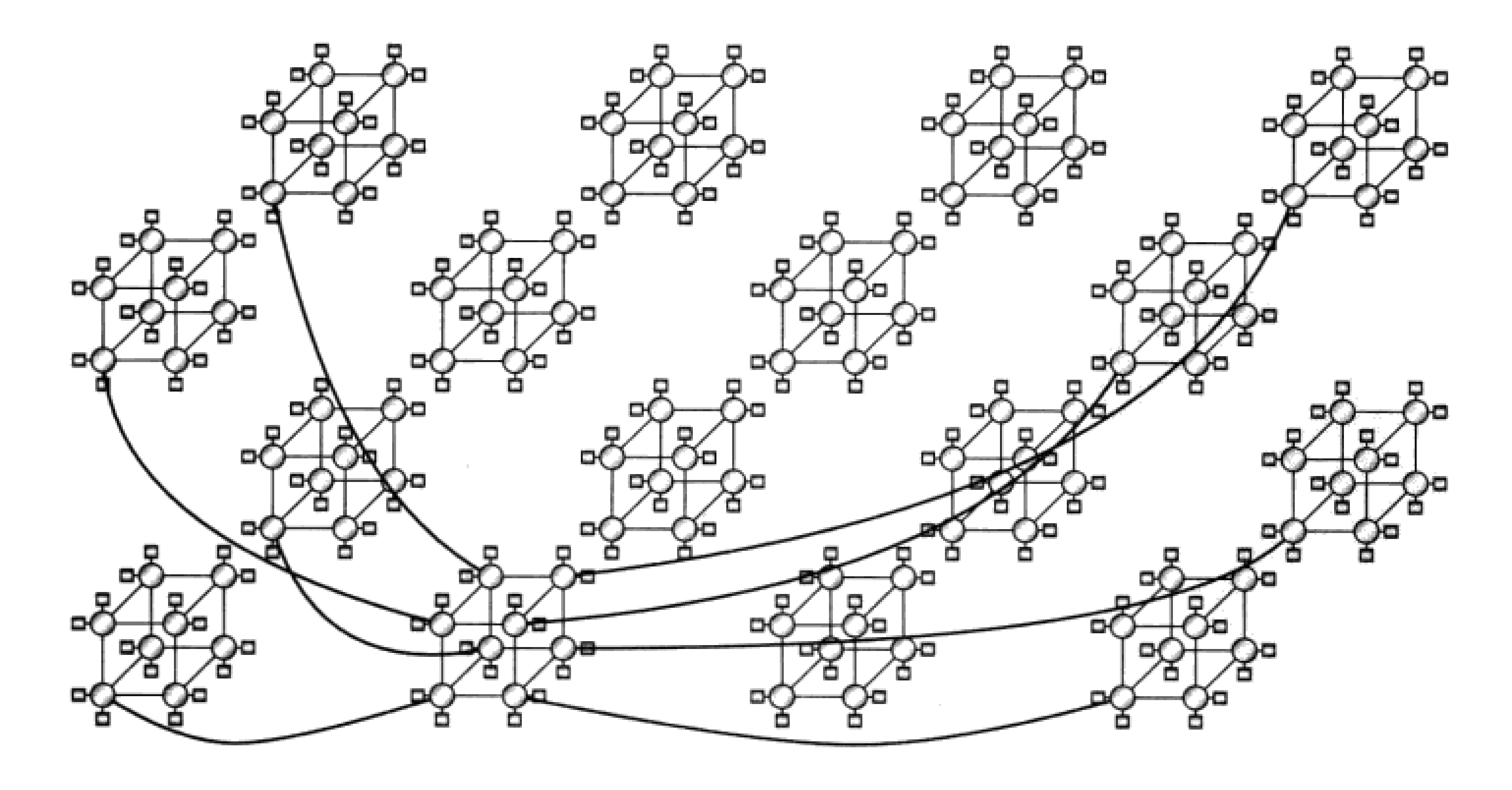


图 16.23 使用 Dual-Cube 构造 Origin2000

16.6.2 Metacube

Dual-Cube 的 Class ID 只有一位。Metacube[14, 17, 20] MC(k, m) 是对 Dual-Cube 的扩充，其地址格式如图 16.24 所示。我们把节点地址的 Class ID 扩充至 k 位，其余共有 2^k 域：m_i，$0 \leqslant i \leqslant 2^k - 1$，每个域有 m 位。因此 MC(k, m) 的节点地址共有 $k + 2^k m$ 位，可连接 $2^{k+2^k m}$ 个节点。

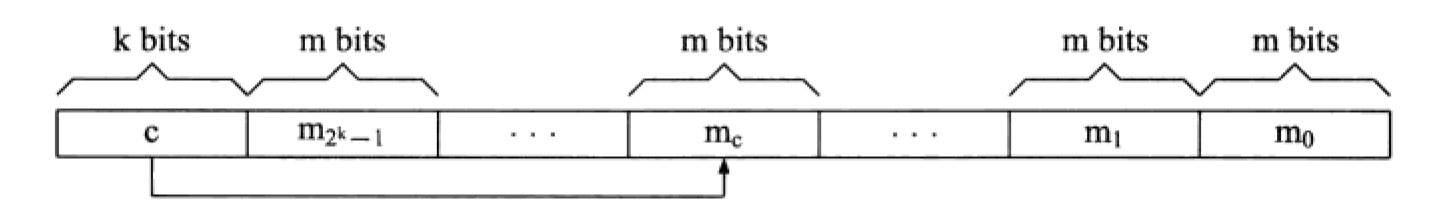


图 16.24 Metacube 地址格式

Metacube 的链路的连接方法如下。在 Class ID 域有 k 条链路。设 Class ID 的值为 c，则在 m_c 域还有 m 条链路。其他域没有链路。因此 MC(k, m) 的节点度为 $k + m$。例如，MC(2, 3) 中节点 (01, 111, 101, 110, 000) 在 k-cube 中的邻接节点为 (0$\underline{0}$, 111, 101, 110, 000) 和 ($\underline{1}$1, 111, 101, 110, 000)；在 m-cube 中的邻接节点为 (01, 111, 101, 11$\underline{1}$, 000)、(01, 111, 101, 1$\underline{0}$0, 000) 和 (01, 111, 101, $\underline{0}$10, 000)。

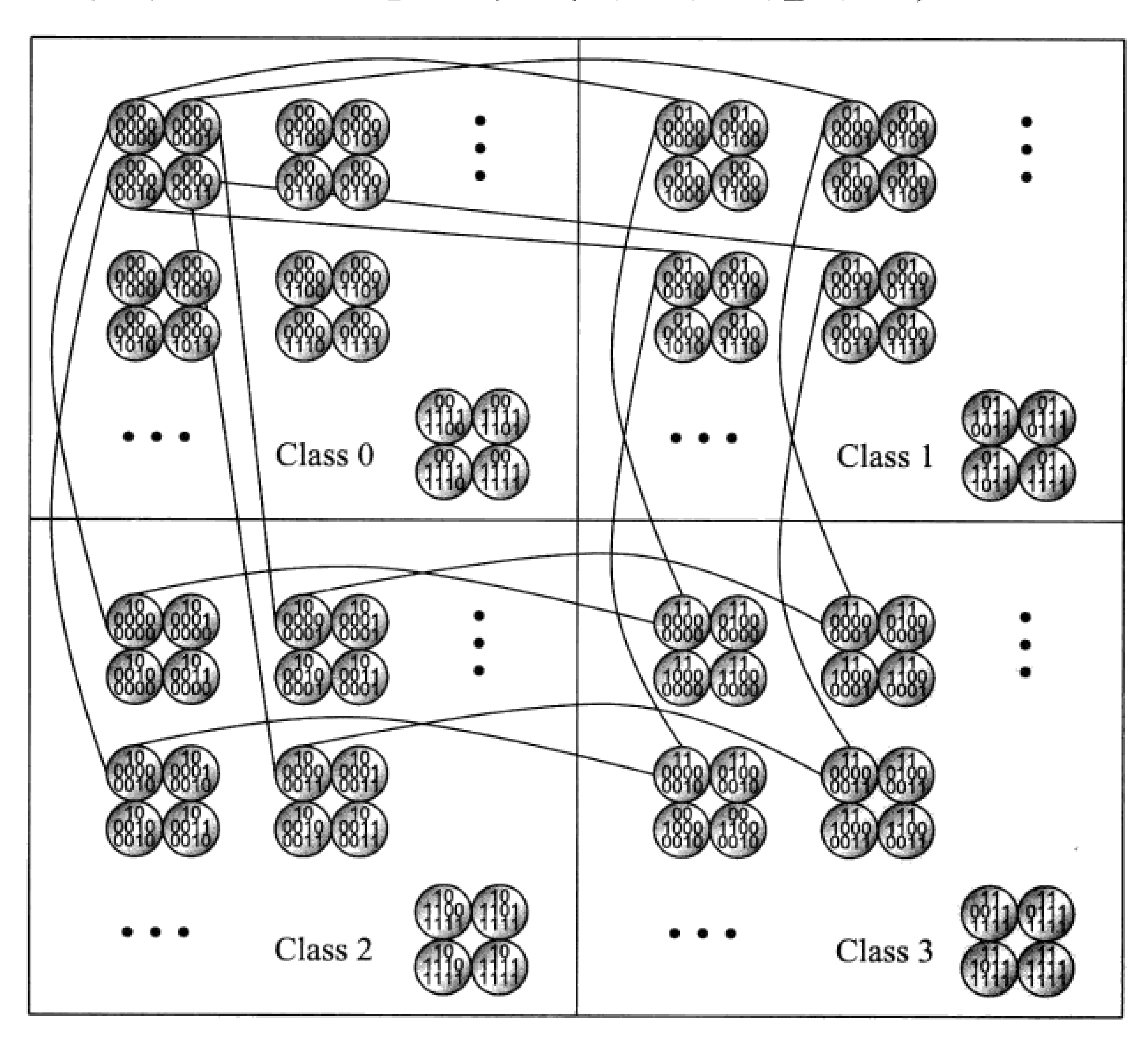


图 16.25 Metacube MC(2,2) 结构示意图

图 16.25 示出的是 MC(2, 2)。由于 MC(2, 2) 有 $2^{2+2^2 \times 2} = 1024$ 个节点，不可能

在图中全部画出，我们只画了一部分节点和链路。

一个 MC(1, m) 就是一个 Dual-Cube DC(m)。我们称 MC(2, m)、MC(3, m) 和 MC(4, m) 分别为 Quad-Cube、Oct-Cube 和 Hex-Cube。而 MC(0, m) 是一个 m-cube。

表 16.2 列出了不同节点度下的 Metacube 节点数，并与 Hypercude 做了比较。从表中我们可知，一个节点度为 6 的 MC(3, 3) 有 $2^{27} = 134\,217\,728$ 个节点，规模应该是足够大了。而节点度为 6 的 Hypercube 只能连接 64 个节点。

表 16.2 Metacube 节点数

节点度	3	4	5	6	7	8
Hypercube	8	16	32	64	128	256
MC(1, m)	32	128	512	2 048	8 192	32 768
MC(2, m)	64	1 024	16 384	2^{18}	2^{22}	2^{26}
MC(3, m)	—	2 048	2^{19}	2^{27}	2^{35}	2^{43}
MC(4, m)	—	—	2^{20}	2^{36}	2^{52}	2^{68}

我们不难证明 MC(k, m) 的直径为 $2^k + 2^k m$，其中 $2^k m$ 是除 Class ID 以外的域的地址位数，2^k 是在 k-cube 中的所谓 Weak-Hamiltonian[14] 的距离。Metacube 的直径比相同规模的 Hypercube 多了 $2^k - k$。由于 $k = 2$ 或 3 就足以构建大规模的系统，因此直径多出的部分不会很大。

同样是 Metacube，我们还可以用另外一种形式来构建，见图 16.26 所示的节点地址格式。我们把 k 位 Class ID 放在最右边，其余分成 m 个域，每个域有 2^k 位。

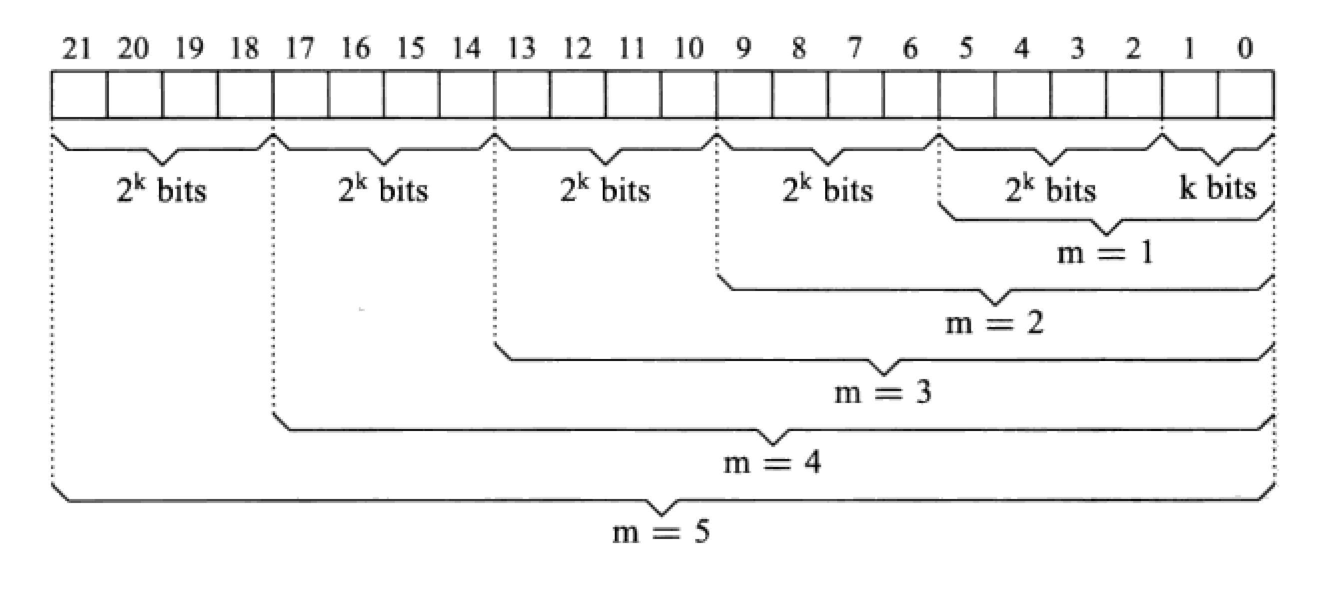


图 16.26 Metacube 另外一种格式的地址

除了 Class ID 域有 k 条链路外，其余 m 条链路分布在 m 个域中，每个域有一条链路，它的位置依 Class ID 的值而定。假设 Class ID 的值为 c，那么 m 条链路所处的位置分别是 $c+k$, 2^k+c+k, $2^k\times2+c+k$, $2^k\times3+c+k$, $\cdots$, $2^k\times(m-1)+c+k$。我们称这种格式的 Metacube 为基于 k-cube 的 Metacube，而称原始的那个为基于 m-cube 的 Metacube。

图 16.27 示出的是一个基于 k-cube 的 MC(2,1)，即 $k = 2$、$m = 1$，基本的 Cluster 是一个 k-cube。从图中看出，这种格式的 Metacube 的链路有比较规整的连接方式，而且在 Cluster 中的节点的地址是连续的。

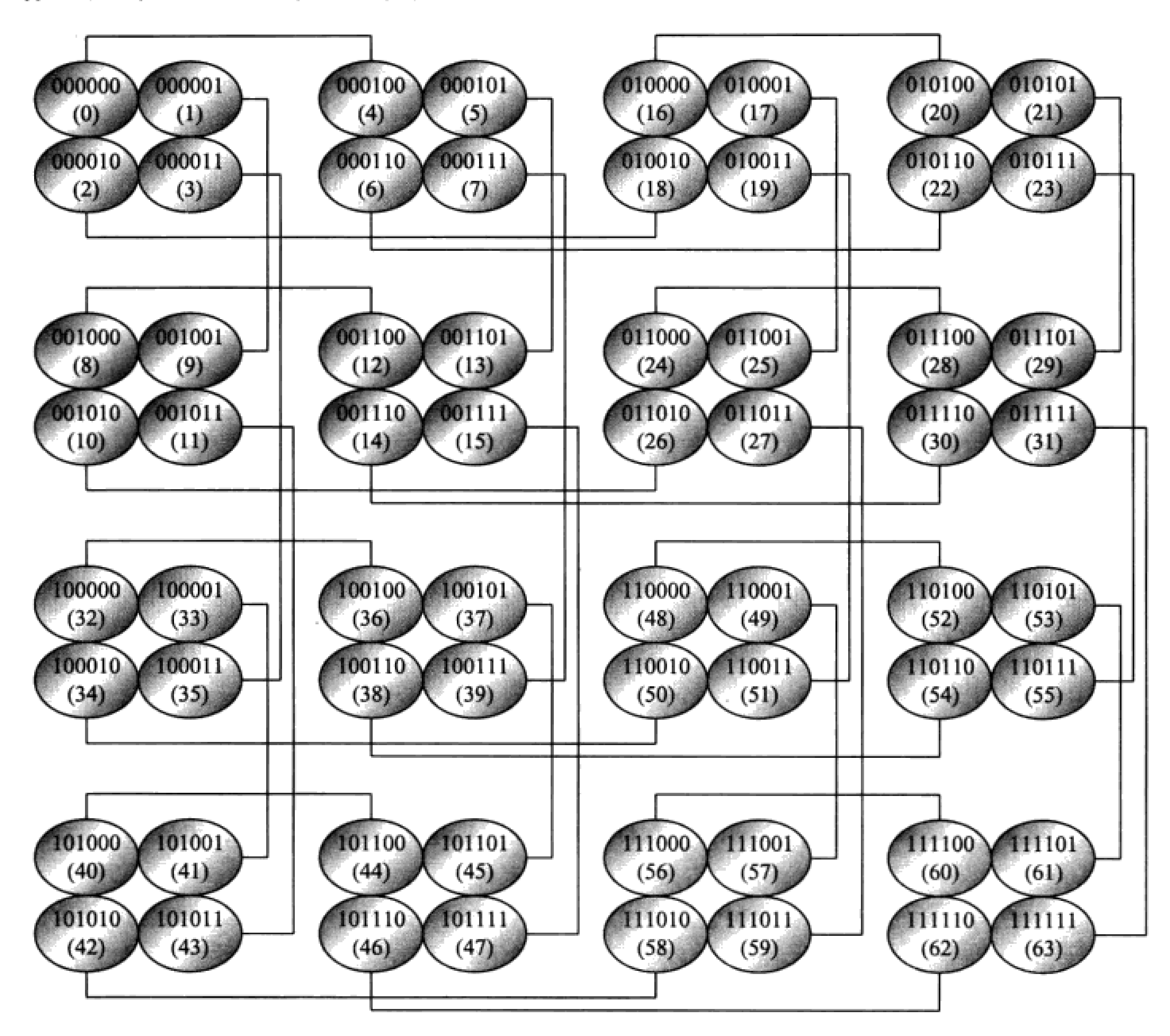


图 16.27 Metacube MC(2,1) 结构图

16.6.3 RDN

RDN (Recursive Dual-Net)[19, 21] 是一种具有递归结构的互联网络。一个 k 级的 RDN 由 $k-1$ 级的 RDN 构建而成。假设 $k-1$ 级的 RDN 有 n 个节点，编号为 $0, 1, \cdots, n-1$，我们称其为 Cluster。一个 k 级的 RDN 有 2n 个 Clusters，每个 Cluster 有 n 个节点。我们把这 2n 个 Clusters 分成两组，每组有 n 个 Clusters，其中的一组定义为类型 0，另外一组定义为类型 1。

如此，一个 k 级的 RDN 的节点地址由 3 部分组成：(c, b, a)，其中 c 等于 0 或 1，用来表示类型；b 的取值范围是 $0 \leqslant b \leqslant n-1$，表示 Cluster 号；a 的取值范围也是 $0 \leqslant a \leqslant n-1$，表示一个 Cluster 内部的节点号。这样，一个 k 级的 RDN 就有 $2n^2$ 个节点。节点之间的第 k 级链路的连接规则是：节点 $(0, b, a)$ 连接到节点 $(1, a, b)$，如图 16.28 所示。

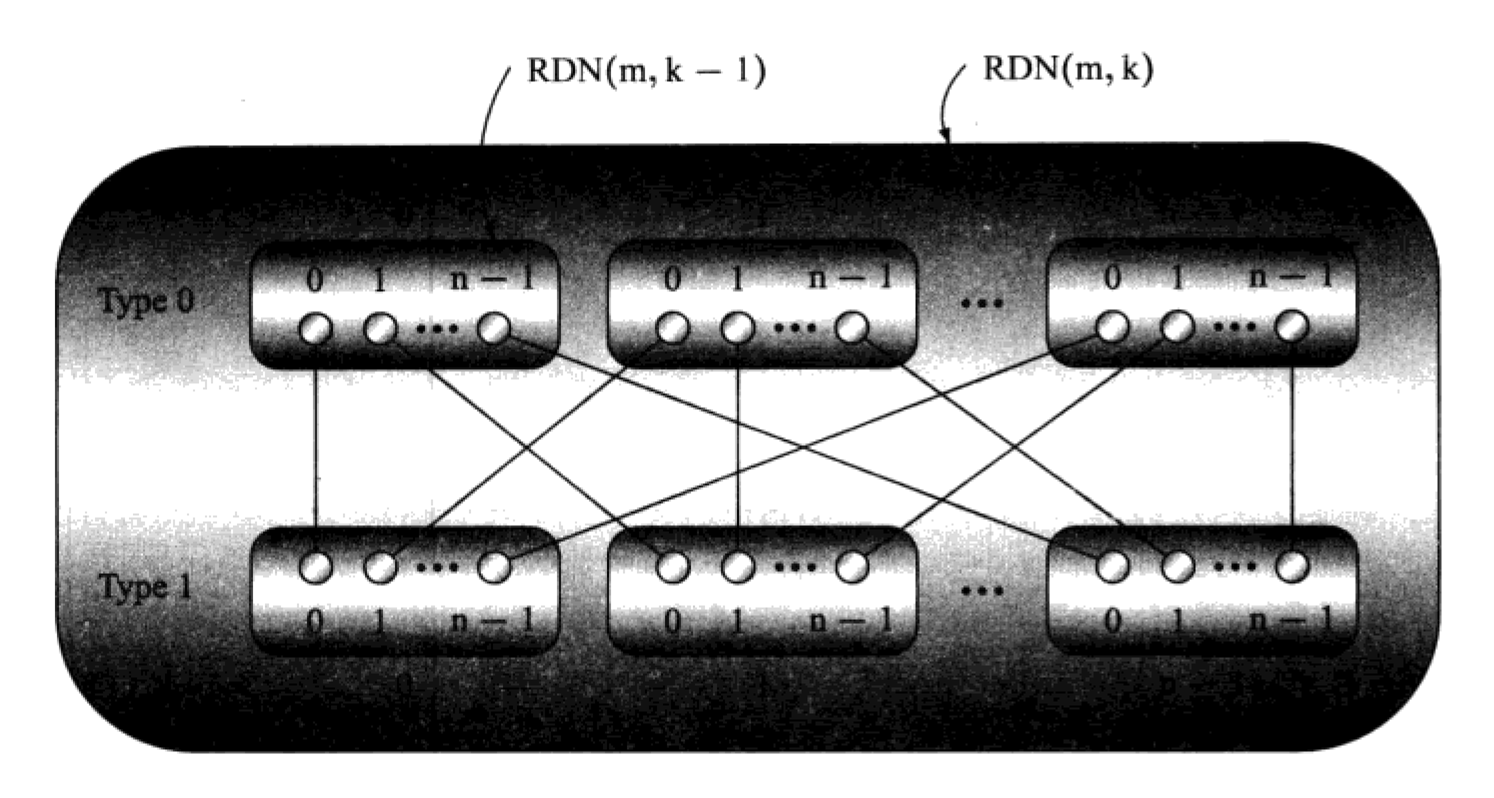


图 16.28 由 k − 1 级的 RDN 构建 k 级的 RDN

三个数字格式的地址可以转换成一个单一数字的地址，然后用同样的方法可以构建一个 $k+1$ 级的 RDN。那么问题来了：一个 0 级的 RDN 是什么呢？答案是任何一种基本的互联网络都可以，我们称其为基础网络 (Base Network)。假设一个基础网络有 m 个节点，节点度为 d，则由此而建成的一个 k 级的 RDN(m, k) 就有 $(2m)^{2^k}/2$ 个节点，节点度为 $d+k$。如果基础网络是对称的，则 RDN 也是对称的。图 16.29 和图 16.30 分别示出了 RDN(4, 1) 和 RDN(4, 2)，其中基础网络是 2-cube。

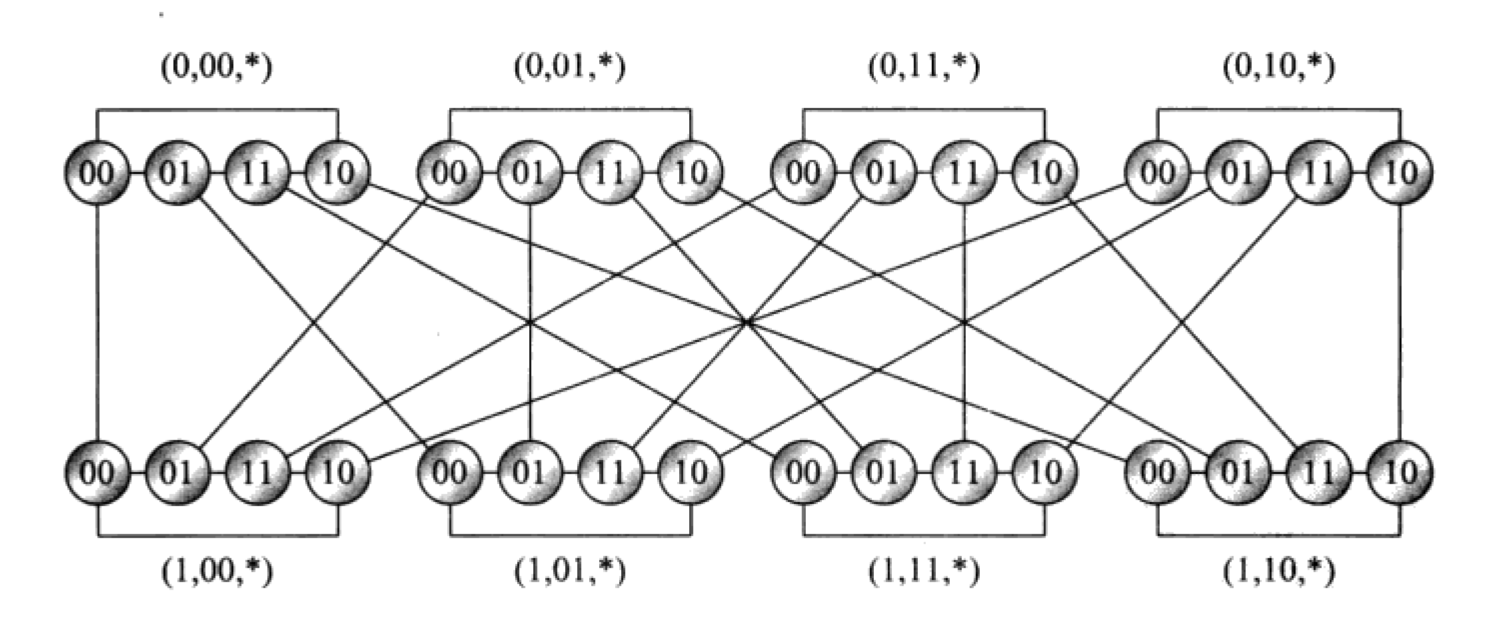


图 16.29 RDN(4, 1) 结构图

假设基础网络的直径为 t，则 RDN(m, k) 的直径为 $2^k t + 2^{k+1} - 2$。如果我们用 5×5 的 2D Torus 作为基础网络，它的节点度是 4，直径是 4。则 RDN(25, 1) 有 $2 \times 25 \times 25 = 1\,250$ 个节点，节点度是 $4+1=5$，直径是 $4+4+2=10$；而 RDN(25, 2) 有 $2 \times 1\,250 \times 1\,250 = 3\,125\,000$ 个节点，节点度是 $5+1=6$，直径是 $10+10+2=22$。

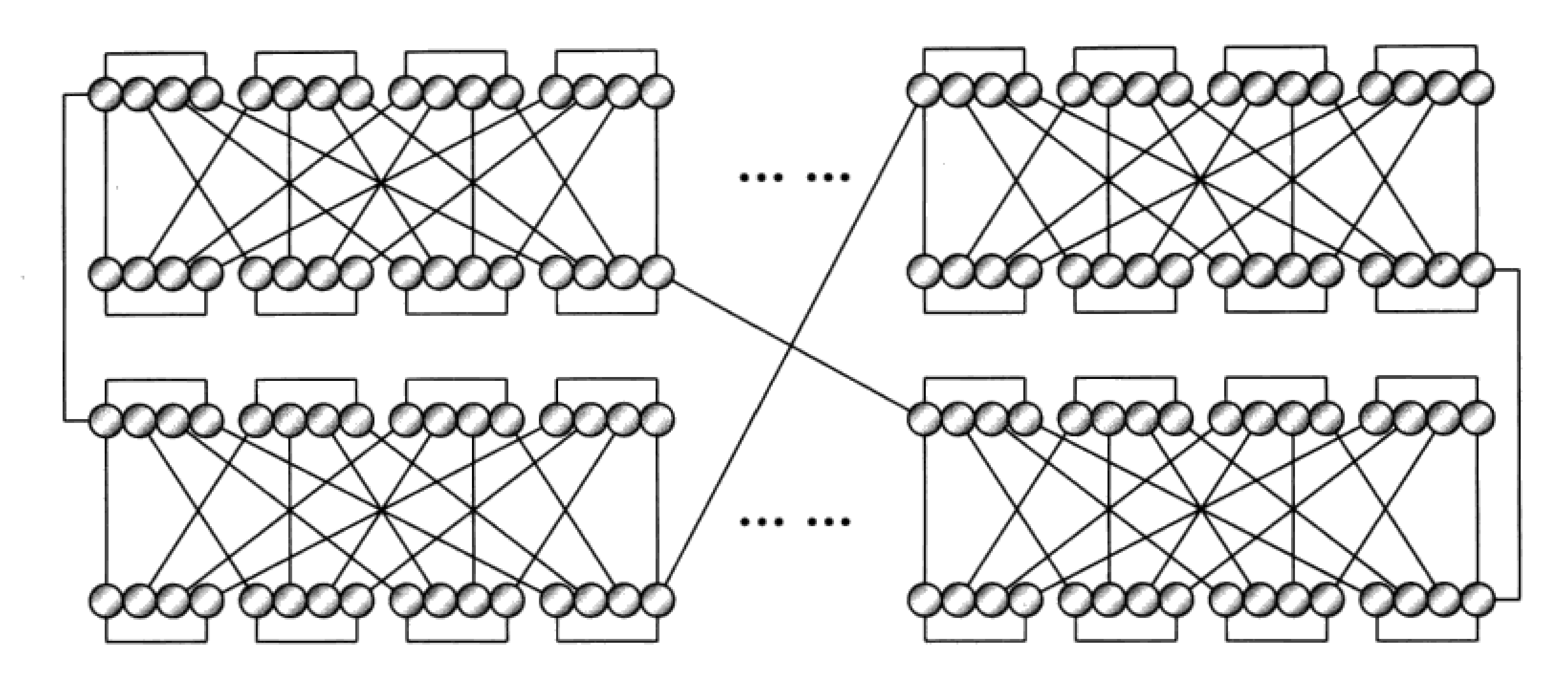


图 16.30 RDN(4, 2) 结构图

我们定义一个参数，称做价格比率 CR (Cost Ratio)，来评价一个对称互联网络的综合价格 (硬件价格和软件价格) 因素。CR 的定义是

$$\mathrm{CR(G)} = \frac{\mathrm{d(G)} + \mathrm{D(G)}}{\log_2 |\mathrm{(G)}|}$$

其中，G 代表一个对称互联网络，$|\mathrm{(G)}|$ 是节点数，d(G) 是节点度，D(G) 是直径。节点度影响互联开关及电缆线的价格 (硬件)，直径影响节点之间的通信时间 (软件)。

表 16.3 小规模和大规模对称互联网络的价格比率 CR

互联网络	节点数	节点度	直径	价格比率
3D-Torus (10)	1 000	6	15	2.11
10-cube	1 024	10	10	2.00
RDN(5^2, 1)	1 250	5	10	1.46
RDN(3^3, 1)	1 458	7	8	1.43
3D-Torus (128)	2 097 152	6	192	9.43
21-cube	2 097 152	21	21	2.00
DC(10)	2 097 152	11	22	1.57
RDN(5^2, 2)	3 125 000	6	22	1.30
RDN(3^3, 2)	4 251 528	8	18	1.18
RDN(5, 3)	50 000 000	5	30	1.37

表 16.3 列出了小规模和大规模互联网络的价格比率 CR，其中 3D-Torus (10) 和 3D-Torus (128) 的结构分别是 $10 \times 10 \times 10$ 和 $128 \times 128 \times 128$；RDN 中的基础网络 5^2、3^3 和 5 分别是 5×5 Torus、$3 \times 3 \times 3$ Torus 和 5 个节点的环，它们右边的数字是

k。从表中我们看出 RDN(3^3, 2) 的价格比率最低：每个节点有 8 条链路，能连接多至 4 251 528 个节点，最长的节点距离仅为 18。

本节描述了三种新型互联网络，它们具有用较少的端口连接更多的 CPU 和存储器板并且保证它们之间的通信距离比较短的优点，非常适合用来构建下一代超大规模的超高性能计算机。

与互联网络有关的其他研究课题包括：(1) 假设 d 为节点度，找出任意两个节点之间的 d 条不相交路径 (Disjoint Paths)；(2) 容错计算 (Fault-Tolerant Computing)，即允许系统中的若干链路或节点出现故障；(3) 构建汉密尔顿环 (Hamiltonian Cycle)，即找出一个环，它连接所有的节点，并且每个节点只在环中出现一次；(4) 算法设计，比如并行前缀计算 (Prefix Computation) 和并行双调排序 (Bitonic Sorting) 等。

16.7 习题

1. 证明 n-cube 的平均距离是 n/2。
2. 把 n-cube 中的一个节点用一个有 n 个节点的环来代替，就构成所谓的 CCC (Cube Connected Cycles)。它的节点度是 3。计算 CCC 的直径和平均距离。
3. 如果在一个互联网络中能找出一个环，它连接了所有的节点，并且每个节点只被连接一次，我们说这个互联网络是 Hamiltonian 的。试证明 Dual-Cube 是 Hamiltonian 的。
4. 证明 RDN(m, k) 的节点数是 $(2m)^{2^k}/2$、直径是 $2^k t + 2^{k+1} - 2$，其中 m 和 t 分别是基础网络的节点数和直径。
5. 试在 Metacube 和 RDN 上实现四种基本的通信操作并分析所需时间。

参 考 文 献

[1] Chu W, Li Y, Cost/performance tradeoff of n-select square root implementations, In: *Australian Computer Science Communications*, vol. 22, pp. 9–16, 2001

[2] Chu W, Li Y, An instruction cache architecture for parallel execution of Java threads, In: *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 180–187, Chengdu, China, Aug. 2003

[3] Hennessy J, Patterson D, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 2006

[4] IDT, *IDT Interprise IDT79RC32355, Integrated Communications Processor RISCore 32300 Family, User Reference Manual*, http://www.idt.com/products/getDoc.cfm?docID=10711, 2003

[5] IEEE, *IEEE 754: Standard for Binary Floating-Point Arithmetic*, http://grouper.ieee.org/groups/754/, 1985

[6] Li Y, Chu W, A performance prediction model for a parallel multithreaded RISC processor architecture, In: *Proceedings of the Sixth IASTED International Conference on Parallel and Distributed Computing and System*, pp. 162–166, Washington DC, USA, Oct. 1994

[7] Li Y, Chu W, The effects of STEF in finely parallel multithreaded processors, In: *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pp. 318–325, North Carolina, USA, Jan. 1995

[8] Li Y, Chu W, Aizup — A pipelined processor design and implementation on Xilinx FPGA chip, In: *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 98–106, Napa, California, USA, April 1996

[9] Li Y, Chu W, A new non-restoring square root algorithm and its VLSI implementations, In: *Proceedings of International Conference on Computer Design – VLSI in Computers and Processors*, pp. 538–544, Austin, Texas, USA, Oct. 1996

[10] Li Y, Chu W, Using computer architecture/organization at the University of Aizu, In: *Workshop on Computer Architecture Education*, San Jose, California, USA, Feb. 1996, http://www.ncsu.edu/wcae/WCAE2/index.html

[11] Li Y, Chu W, Implementation of single precision floating point square root on FPGAs, In: *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 226–232, Napa, California, USA, April 1997

[12] Li Y, Chu W, Parallel-array implementations of a non-restoring square root algorithm, In: *Proceedings of International Conference on Computer Design – VLSI in Computers and Processors*, pp. 690–695, Austin, Texas, USA, Oct. 1997

[13] Li Y, Peng S, Chu W, Efficient communication in Metacube: A new interconnection network, In: *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 165–170, Manila, Philippines, May 2002

[14] Li Y, Peng S, Chu W, Metacube — A new interconnection network for large scale parallel systems, *Australian Computer Science Communications*, vol. 24, pp. 29–36, 2002

[15] Li Y, Peng S, Chu W, Efficient collective communications in Dual-Cube, *The Journal of Supercomputing*, vol. 28, no. 1, pp. 71–90, April 2004

[16] Li Y, Peng S, Chu W, MCore: A simple structure for effective overlay multicast on mobile ad hoc networks, In: *Proceedings of The IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 341–346, Dallas, USA, Nov. 2006

[17] Li Y, Peng S, Chu W, A new presentation of Metacube for algorithmic design and case studies: Parallel prefix computation and parallel sorting, *Journal of Chinese Institute of Engineer*, vol. 32, pp. 939–950, Nov. 2009

[18] Li Y, Peng S, Chu W, Optimal algorithms for finding a trunk on a tree network and its application, *The Computer Journal*, vol. 52, no. 2, pp. 268–275, March 2009

[19] Li Y, Peng S, Chu W, Recursive Dual-Net: A new universal network for supercomputers of the next generation, In: *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, Springer, Lecture Notes in Computer Science, pp. 809–820, Taipei, Taiwan, June 2009

[20] Li Y, Peng S, Chu W, Metacube — A versatile family of interconnection networks for extremely large-scale supercomputers, *The Journal of Supercomputing*, vol. 53, no. 2, pp. 329–351, August 2010

[21] Li Y, Peng S, Chu W, Parallel prefix computation and sorting on a Recursive Dual-Net, *The Journal of Information Processing Systems*, vol. 7, no. 2, pp. 43–59, June 2011

[22] MIPS, *MIPS32 Architecture For Programmers Volume I: Introduction to the MIPS32 Architecture*, MIPS Technologies, Inc., 2001

[23] MIPS, *MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set*, MIPS Technologies, Inc., 2001

[24] MIPS, *MIPS32 Architecture For Programmers Volume III: The MIPS32 Privileged Resource Architecture*, MIPS Technologies, Inc., 2001

[25] Pagiamtzis K, Sheikholeslami A, Content-addressable memory (CAM) circuits and architectures: A tutorial and survey, *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006

[26] Patt Y, Patel S, *Introduction to Computing Systems: From Bits and Gates to C and Beyond (2nd edition)*, McGraw-Hill Science Engineering, 2003

[27] Patterson D, Hennessy J, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers Inc., 2008

[28] PCISIG, *PCI Local Bus Specification, Revision 2.3*, PCI Special Interest Group, March 28, 2002

[29] Schwarz E, Sigal L, McPherson T, CMOS floating-point unit for the S/390 parallel enterprise server G4, *IBM Journal of Research and Development*, vol. 41, no. 4/5, pp. 475–488, July/Sep. 1997

[30] Philips Semiconductors, *The I2C-Bus Specification*, NXP, http://www.nxp.com/acrobat_download2/literature/9398/39340011.pdf, Jan. 2000

[31] IEEE Computer Society, *IEEE Standard Verilog Hardware Description Language, IEEE Std. 1364-2001 (Revision of IEEE Std 1364-1995)*, http://ieeexplore.ieee.org/iel5/7578/20656/00954909.pdf?arnumber=954909, 2001

[32] Suzuki S, *The Digital Circuit Design*, (in Japanese), CQ Press, 2006

[33] Sweetman D, *See MIPS Run*, Morgan Kaufmann Publishers Inc., 2006

[34] Watanabe K, Chu W, Li Y, Exploiting Java instruction/thread level parallelism with horizontal multithreading, In: *Australian Computer Science Communications*, vol. 23, pp. 122–129, 2001

[35] 李三立，李亚民，RISC —— 单发射与多发射体系结构，北京：清华大学出版社，1993

[36] 李亚民，计算机组成与系统结构，北京：清华大学出版社，2000

[37] 朱子玉，李亚民，CPU 芯片逻辑设计技术，北京：清华大学出版社，2005

图 索 引

表 索 引

术 语 索 引

［Ｇｅｎｅｒａｌ Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝计算机原理与设计 Ｖｅｒｉｌｏｇ ＨＤＬ版
作者＝李亚民著
页数＝５２０
出版社＝北京市：清华大学出版社
出版日期＝２０１１．０６
ＳＳ号＝１２８５２５６９
ＤＸ号＝０００００８１５０９０７
ＵＲＬ＝ｈｔｔｐ：／／ｂｏｏｋ．ｓｚｄｎｅｔ．ｏｒｇ．ｃｎ／ｂｏｏｋＤｅｔａｉｌ．ｊｓｐ？ｄｘＮｕｍｂｅｒ＝０００００８１５０９０７＆ｄ＝６３９８３５０Ｅ５０Ｃ８６６ＣＣ９５３３１５９０３ＡＡ０Ｆ０３０